Universidad Complutense de Madrid

Facultad de Informática

Proyecto de Sistemas Informáticos

Generación de casos de prueba para SQL

José Luzón Martín y Antonio Tenorio Fornés

Directora: Sonia Estévez Martín

Codirector: Rafael Caballero Roldán

Junio 2012

Agradecimientos

Queremos agradecer a todos los que han hecho posible este proyecto. A Rafael Caballero por idear el proyecto, por apoyarnos en todo momento y por creer en nosotros impulsando la publicación de nuestras propuestas surgidas del desarrollo del proyecto. A Sonia Estévez por ayudarnos a dar forma al proyecto y haber accedido a dirigirlo, aún sin estar directamente relacionado con sus líneas de investigación. A Ignacio Salcedo por el modelo de restricciones sobre strings que podremos utilizar en las siguientes versiones de la herramienta. Y a nuestras familias y parejas que han sufrido nuestra ausencia física y mental debida al desempeño con el que hemos afrontado este reto.

¡GRACIAS!

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado.

José Luzón Martín Antonio Tenorio Fornés

Abstract

The use of databases is widely extended, however, there is a lack of testing and debugging tools to assist it's development. To test the correctness of an SQL query is difficult using actual databases, either because the number of rows in populated databases is usually large or because they have few data in early stages of development.

To solve this problem we have developed a tool that, given an SQL database definition and a target view, finds a database instance for which the view's query does not return an empty answer, this is considered to be a positive test case. Implementing recent research results, we transform the search for this instance to a constraint satisfaction problem. We also expands this technique's scope by presenting a new approach for nested existential conditions.

We divide the test case generation process in three phases. First the SQL code of the database is analysed. Then, given the target view's name, the formulas representing the conditions that an instance has to satisfy in order to be a test case for this view are generated. Finally this formulas are translated into an specific constraint language, which solutions are the desired test cases.

This work presents a prototype which introduces new approaches in the field of databases and shows a way of solving the lack of test tools for it's development.

Keywords

Constraints, SQL, Test Case, Test, Databases

Resumen

El uso de bases de datos está ampliamente extendido, sin embargo, hay una falta de herramientas de depuración y prueba que asistan su desarrollo. Comprobar la corrección de una consulta SQL es difícil usando bases de datos reales, ya sea porque el número de filas en bases de datos pobladas es normalmente muy elevado o por la carencia de datos en las primeras fases del desarrollo.

Para solventar este problema hemos desarrollado una herramienta que, dada la definición de una base de datos SQL y una vista objetivo, encuentra una instancia de la base de datos para la cual la consulta de la vista no devuelve una respuesta vacía, lo que se considera un caso de prueba positivo. Aplicando recientes resultados de investigación, transformamos la búsqueda de esta instancia a un problema de resolución de restricciones. Ampliamos también el ámbito de aplicación de esta técnica presentando un nuevo enfoque para tratar condiciones existenciales anidadas.

Dividimos el proceso de obtención de casos de prueba en tres fases. Primero se analiza el código SQL que define la base de datos a estudiar. Posteriormente, dado el nombre de la vista objetivo, se generan las fórmulas que representan las condiciones que una instancia debe cumplir para ser un caso de prueba de dicha vista. Finalmente se traducen estas fórmulas a un lenguaje de programación con restricciones específico, cuyas soluciones constituyen los casos de prueba buscados.

El trabajo presenta un prototipo novedoso en el campo de las bases de datos y muestra una forma de suplir la carencia actual de herramientas de prueba para su desarrollo.

Palabras clave

Restricciones, SQL, Caso de Prueba, Prueba, Bases de Datos

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Definición del problema

El uso de bases de datos está ampliamente extendido en todo tipo de aplicaciones informáticas. Sin embargo, el desarrollo de bases de datos relacionales es una tarea compleja para la que no existe apenas herramientas complementarias de depuración y prueba [1][2]. Comprobar si una consulta es correcta es difícil utilizando instancias reales, ya sea porque el número de filas en las tablas suele ser muy elevado en bases de datos pobladas o por que haya pocos datos en la fase inicial de desarrollo. Por eso, es importante disponer de instancias significativas.

Para dar solución a este problema hemos desarrollado una herramienta que permite encontrar una instancia de la base de datos para la cual una determinada consulta devuelve una respuesta no vacía, lo que se denomina un caso de prueba positivo. Siguiendo lo propuesto en [3], reducimos el problema de encontrar un caso de prueba a un problema de resolución de restricciones [4]. Así, a partir del estudio de la definición del esquema de la base de datos y dado el nombre de la vista o tabla de la cual se quiere obtener el caso de prueba y una cota superior del tamaño que tendrán las tablas del caso de prueba relacionadas con dicha vista o tabla podemos obtener los valores que conforman dicho caso de prueba.

1.2. Restricciones y Casos de Prueba

La programación con restricciones es un paradigma de programación que utiliza condiciones y relaciones entre variables simbólicas que acotan los posibles valores que pueden tomar estas. Las restricciones son una forma de expresar estas condiciones y relaciones mediante fórmulas de la lógica de primer orden cuyas variables tienen un dominio predefinido. Se considera una instancia como solución valida si los valores concretos asignados a las variables simbólicas satisfacen estas fórmulas. Expresando las sentencias del lenguaje de definición de la base de datos como restricciones sobre las variables de una instancia simbólica

de la base de datos podemos encontrar soluciones concretas que conforman casos de prueba. Para ello, desde el estudio del código de la base de datos y aplicando las técnicas propuestas en [3] y extendidas en [5] generamos una representación propia de las restricciones que luego traducimos a lenguajes de restricciones concretos como *MiniZinc* [6] para su resolución.

La utilización de casos de prueba en el desarrollo de software es una práctica ampliamente extendida. Como su propio nombre sugiere, un caso de prueba es una herramienta para la comprobación del correcto funcionamiento de componentes software.

Dentro de los paradigmas de pruebas de software podemos encontrar *Black-Box Testing* y *White-Box Testing*.

- Black-Box Testing es un método de pruebas de software que sigue el *modelo de caja negra*, en el cual simplemente se dan unos datos de entrada a la aplicación y se obtienen unos resultados. En este método no es necesario conocer el código del programa para poder comprobar su funcionamiento, únicamente basta con conocer la respuesta que el programa debe proporcionar ante los datos sumistrados. Si la respuesta del programa es la esperada, entonces se considera que el caso de prueba es positivo, es decir, se cumplen los requisitos de funcionalidad. En otro caso, el caso de prueba se dice que es negativo.

- White-Box Testing, o *modelo de caja blanca*, es un método de pruebas de software en el cual, a diferencia de Black-Box Testing, se tiene una visibilidad completa del funcionamiento interno del programa, de su lógica y estructura. En este modelo se tiene acceso al código del programa, permitiendo realizar una comprobación más exhaustiva del mismo. La primera idea que puede plantearse es comprobar que todas y cada una de las posibles rutas que puede tomar la ejecución de un programa se comportan como se espera. Por ejemplo, suponiendo que nos encontramos un código escrito en un lenguaje tipo C, es muy habitual encontrarse con la siguiente situación:

  ```
  if(condicion) {
      // cuerpo del if
  } else {
      // cuerpo del else
  }
  ```

  En este caso, la ejecución del código puede tomar dos rutas, dependiendo de si se cumple la condición de la sentencia `if-else` o no. En caso de que la evaluación de la condición sea cierta, se ejecutará el código correspondiente al cuerpo de la sección `if` de la sentencia. Por el contrario, si la evaluación de la condición resulta falsa, se ejecutará el cuerpo de la sección `else` de la sentencia. Por tanto, utilizando el modelo de caja blanca para probar este código, se tienen dos casos de prueba bien diferenciados:

1. Uno que haga que la evaluación de la condición sea cierta, ejecutándose así la ruta del `if`.
2. Otro que haga que el resultado de evaluar la condición sea falsa, por lo que se ejecutará la ruta del `else`.

Para una simple sentencia `if-else` tenemos, al menos, dos casos de prueba. Sin embargo en sistemas software de carácter industrial o profesional, realizar un caso de prueba para cada posible ruta que pueda tener la ejecución del código es una tarea muy costosa. Por ello, se utilizan diversos *criterios de recubrimiento* [7] del código del programa. Estos criterios de recubrimiento pueden ayudar, junto con otras técnicas de desarrollo de casos de prueba (test de unidad, test de integración, etc), a la obtención de un conjunto de casos de prueba que cubra los casos más significativos. El hecho de disponer de un conjunto de casos de prueba significativos, en función del criterio de recubrimiento seguido, hace que dicho conjunto sea tratable. Por el contrario, obtener un conjunto de casos de prueba que cubra todas y cada de las posibles combinaciones de valores que pueden tomar las variables del programa, generalmente no es computable, pues nunca podemos estar seguros de haber recorrido todos los posibles caminos que puede tomar un programa general. Por ello, detectar los posibles puntos débiles de un sistema software puede ser de gran ayuda en la elección de los criterios de recubrimiento a utilizar.

En el caso particular de SQL, un buen recubrimiento consiste en transformar una consulta SQL inicial en un conjunto de consultas independientes, de forma que cada una de estas consultas compruebe condiciones individuales de la consulta inicial [2] (*full predicate coverage* en la literatura). De esta forma, se puede obtener un conjunto de casos de prueba para cada una de estas consulta. La unión de estos conjuntos forma un conjunto de casos de prueba para la consulta original.

Nuestro trabajo se centra en la generación de casos de prueba a partir de un conjunto de tablas y de vistas SQL y no realiza el proceso de transformación de la vista inicial en el conjunto de casos de prueba antes descrito, sin embargo, nuestro puede ser utilizado para encontrar dichos casos de prueba si se le proporcionan las vistas que realizan el recubrimiento mencionado.

1.3. Aportaciones

El presente proyecto supone la implementación de las ideas presentadas en [3] y la extensión de las mismas para tratar con subconsultas existenciales anidadas y su negación [5]. Las subconsultas existenciales en SQL se declaran mediante la palabra reservada `EXISTS`, y juegan un papel muy importante a la hora de la definición de vistas SQL. Presentamos un prototipo que podría llegar a ser de gran ayuda en el desarrollo de bases de datos, sobre todo en fase de depuración y pruebas (ver C.1.1 para obtener el prototipo). Cabe decir que dicho prototipo es limitado en cuanto a tipos de datos soportados y la gramática SQL que acepta, aunque tiene las bases para que pueda ser extendido en estos aspectos. La mejora

de estas características constituye parte del trabajo futuro a corto plazo de la herramienta y supondría una mejora sustancial de su utilidad práctica.

1.4. Estructura del trabajo

El resto de esta memoria está estructurada en tres grandes capítulos: **Análisis y diseño**, en el cual se hace especial énfasis en el análisis del problema definido en la sección 1.1, y se realiza un planteamiento de como decidimos que debía ser la herramienta; **Desarrollo e implementación**, que realiza una visión general del proceso de desarrollo de la herramienta, y se profundiza en la implementación de la misma; **Conclusiones**, donde exponemos el trabajo realizado, así como lo que nos ha aportado en nuestra formación como ingenieros y el trabajo que podría complementar nuestra herramienta en versiones futuras.

Capítulo 2

Análisis y diseño

En este capítulo se estudian las características del problema que queremos resolver y se expone la arquitectura encontrada para ello. Primero se exponen y justifican las diferentes decisiones de diseño que se han adoptado. Posteriormente se indica el ámbito de aplicación del proyecto. Finalmente se hace un breve análisis de los módulos fundamentales del proyecto.

2.1. Decisiones de diseño

En esta sección se exponen las decisiones más relevantes que se tomaron en el diseño de la herramienta. Estas decisiones guiaron el desarrollo, dotándole de una arquitectura y acotando las tecnologías que se usarían para ello.

- *Pipeline*

  El prototipo se diseño en forma de *pipeline* (ver sección 2.3), es decir, se dividió en módulos funcionales de forma que la salida de un módulo es la entrada del siguiente. Los principales módulos identificados son el analizador sintáctico o parser, el generador de fórmulas y el traductor a código de programación con restricciones o de resolutor específico. Esto nos facilita la depuración y las pruebas a la par que refuerza la modularidad del proyecto. Permitiendo la reutilización y sustitución de módulos así como la ampliación de las posibilidades de la herramienta, realizando diferentes implementaciones para cada unidad funcional del proyecto.

- *Resolutor*

  En una fase inicial se decidió utilizar Gecode [8] como resolutor de las restricciones generadas. Posteriormente se cambió la decisión por MiniZinc[6], un lenguaje con mayor nivel de abstracción que facilitó el desarrollo y que permite la elección del resolutor a utilizar.

- *Restricciones*

Utilizamos una representación interna de las restricciones, lo que nos permite implementar módulos que las traduzcan a distintos lenguajes y resolutores.

- *Parser*

  El analizador sintáctico estudia el código de definición de la base de datos y genera la representación de la misma. Nos basamos en la gramática de SQL-92 propuesta en [9] para implementarlo. Se decidió usar Flex [10] y Bison [11] como analizadores léxicos y sintácticos respectivamente.

- *Lenguaje de programación*

  El lenguaje de programación utilizado es C++. Esta decisión estuvo influenciada por la elección de Gecode como resolutor ya que éste es una librería de C++. Además, el equipo de desarrollo estaba familiarizado con el lenguaje.

- *Salida de datos*

  Presentamos el caso de prueba encontrado como sentencias de inserción SQL, esto facilita la comprensión y la inclusión directa de los resultados en la base de datos.

2.2. Ámbito de aplicación

El proyecto ha sido diseñado para soportar las principales características semánticas de las consultas y sentencias de creación de vistas y tablas SQL. Así, se ha seleccionado un subconjunto representativo de estas sentencias SQL para las cuales nuestro sistema resuelve el problema de generar casos de prueba. Dando soluciones a este lenguaje SQL reducido damos también solución a gran parte del lenguaje SQL no soportado directamente ya que muchas de las sentencias que no están contempladas pueden ser transformadas a sentencias equivalentes admitidas por nuestro sistema (ver apéndice D). Consideramos como trabajo futuro la extensión de la herramienta para soportar una mayor variedad de expresiones SQL.

Nuestra herramienta admitirá código SQL de definición de bases de datos con las siguientes características:

- **Claves** primarias y ajenas

  `PRIMARY KEY, FOREIGN KEY`

- **Uniones e intersecciones** de consultas

  `UNION, INTERSECTION`

- **Expresiones aritmético-lógicas** en las secciones FROM y SELECT

- **Expresiones lógicas**

  `AND, OR, NOT, =, <, <=, =>,<>`

- **Expresiones aritméticas**

  `+, -, /, *`

- **Tipos** enteros no nulos

  `INT, INTEGER`

- **Consultas básicas** de la forma `SELECT` E1 ... En `FROM` R1 ... RM `WHERE` C con posibles renombramientos de las expresiones de la clausula `SELECT` (E1 ...En) y las relaciones de la clausula `FROM` (R1 ... Rm)

- **Consultas existenciales** anidadas. El hecho de admitir este tipo de consultas dota a nuestra herramienta de gran versatilidad en cuanto lenguaje SQL admitido, y su tratamiento para la generación de restricciones resulta de gran importancia [5].

  `EXISTS, NOT EXISTS`

- **Consultas con agregados**[1]

  `GROUP BY, HAVING`

Además el código admitido tendrá las siguientes restricciones:

- **Clausula `FROM` sin subconsultas**. Esto no supone una pérdida de generalidad pues las subconsultas en el `FROM` pueden ser sustituidas por vistas, admitidas en nuestra herramienta.

- `DISTINCT` **no permitido**. El operador `DISTINCT` puede ser sustituido por consultas con agregados equivalentes.

- **Operador `MINUS`, operaciones `JOIN` y valores nulos no considerados**. Por simplicidad no se consideran estas características, aunque podrían ser incluidas en futuras implementaciones

- **Consultas recursivas** no contempladas en la herramienta.

El diseño del prototipo permite además la inclusión de nuevos tipos de datos.

2.3. Estructura en "pipeline"

Nuestro proyecto se compone de distintos módulos funcionales que se relacionan de forma que la salida de uno de esos módulos es la entrada del siguiente. Esta estructura se denomina *pipeline* (figuras 2.1 2.2). Esto nos proporciona una mayor flexibilidad, pudiendo añadir o sustituir diferentes módulos para cada una de las principales tareas de nuestro sistema y permitiendo la utilización de cada uno de los módulos como sistemas completos independientes.

[1] La versión actual del prototipo no soporta aún esta característica.

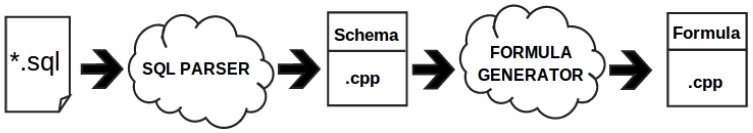

Figura 2.1: Pipeline del prototipo (primera parte).

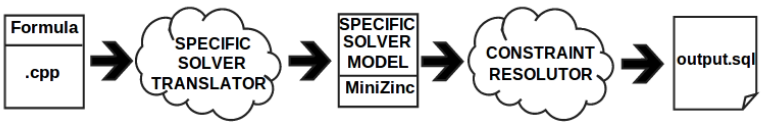

Figura 2.2: Pipeline del prototipo (segunda parte).

2.3.1. Analizador sintáctico

El estudio de la base de datos de la cual se quieren obtener los casos de prueba se hace analizando sintácticamente las sentencias SQL que la definen. La herramienta acepta un subconjunto de SQL (ver apéndice A) convirtiendo estas sentencias en una estructura de objetos de C++ que es utilizada en los siguientes pasos para generar las restricciones (sección 2.3.2).

Utilizamos Flex [10] y Bison [11] como analizador léxico y sintáctico respectivamente.

El analizador léxico nos permite traducir conjuntos de caracteres de la entrada de texto en *tokens* dotando de significado dichos caracteres. Algunos *tokens* tienen asociado un valor de un tipo de datos, por ejemplo la cadena de caracteres *2012* será un token identificado como entero y con un valor entero asociado 2012.

El analizador sintáctico obtiene del analizador léxico una secuencia de *tokens*. A partir de ella se obtiene la estructura y se construye el modelo de la base de datos definida en la entrada de texto analizada.

La gramática utilizada (ver apéndice A) es una adaptación de la gramática de SQL-92 propuesta en [9]. Es importante notar que, aunque nuestra herramienta contenga estos elementos de análisis de sentencias SQL, no es un compilador, por lo que no dará respuestas significativas al usuario si este introduce una expresión SQL mal formada.

2.3.2. Generación de fórmulas

La principal función del proyecto es la transformación del problema de encontrar un caso de prueba positivo para una determinada vista SQL en un prob-

lema de resolución de restricciones. Para ello seguimos las ideas presentadas en [3] y ampliamos su ámbito de aplicación [5] para poder usarla con subqueries existenciales anidadas.

Para representar las restricciones obtenidas usamos una representación interna (figura 3.3), lo que nos permite traducir a una gran variedad de lenguajes de restricciones para su resolución.

A continuación presentamos el método utilizado para transformar sentencias SQL en restricciones.

Transformación a restricciones

En la herramienta, el usuario indica la vista SQL de la que quiere obtener el caso de prueba así como el máximo número de filas del caso de prueba para cada tabla relacionada con dicha vista. A partir de estos datos, se genera una instancia simbólica de estas tablas (con variables simbólicas frescas para cada atributo de cada fila) a la que posteriormente se aplicarán las restricciones generadas para obtener, a través de la resolución de las mismas, una instancia concreta que represente un caso de prueba positivo de la vista objetivo.

Siguiendo la metodología propuesta en [3][5], para cierta instancia simbólica d de la base de datos D, representamos las restricciones asociadas a cada relación (tabla o vista) R como un multiconjuto $\theta(R, d)$ de pares (ψ, u) donde ψ es una fórmula de primer orden y u una fila de dicha relación. Representamos estas filas como sustituciones de sus atributos por expresiones sobre variables simbólicas y constantes con la siguiente notación: $(R.A_1 \mapsto e_1, \ldots, R.A_m \mapsto e_m)$, siendo $R.A_1, \ldots, R.A_m$ los atributos de la relación y $e_1, \ldots, e_m$ expresiones de variables simbólicas y constantes. Definimos el multiconjunto $theta(R, d)$ como:

1. Para cada tabla T con atributos $T.C_1, \ldots, T.C_m$ de D creamos una instancia simbólica d con filas $\mu_1, \ldots, \mu_n$ (donde n es el número de filas especificado por el usuario) y donde cada fila μ_i está compuesta por variables simbólicas frescas X_{ij} con j $\in 1 \ldots m$, siendo el tipo de X_{ij} igual al tipo del atributo $T.C_j$.

   - Si la definición de T no tiene claves primarias (`PRIMARY KEY`) ni ajenas (`FOREIGN KEY`): $\theta(T, d) = \{\!|(true, \mu_1), \ldots, (true, \mu_n)|\!\}$. Donde $true$ simboliza que no hay restricciones sobre las filas μ_i.
   - Si T tiene una clave primaria para las columnas $T.C_1, \ldots, T.C_p$: Sea T' la tabla T sin dicha clave primaria. Supongamos que $\theta(T', d) = \{\!|(\psi_1, \mu_1), \ldots, (\psi_n, \mu_n)|\!\}$, entonces $\theta(T, d)$ es igual a

   $$\{\!|((\psi_i \wedge (\bigwedge_{j=1, j\neq i}^{n} (\bigvee_{k=1}^{p} \mu_i(T.C_k) \neq \mu_j(T.C_k)))), \mu_i) | i \in 1, \ldots, n|\!\}$$

   - Si T tiene una clave ajena para las columnas $T.C_1, \ldots, T.C_f$ referenciando las columnas $T2.C'_1, \ldots T2.C'_f$ de la tabla $T2$: Sea T' la tabla T

sin la clave ajena. Supongamos que $\theta(T', d) = \{\!|(\psi_1, \mu_1), \ldots, (\psi_n, \mu_n)|\!\}$ y que $d(T_2) = \{\!|\nu_1, \ldots, \nu_{n'}|\!\}$, entonces $\theta(T, d)$ es igual a

$$\{\!|((\psi_i \wedge (\bigvee_{j=1, j\neq i}^{n'} (\bigwedge_{k=1}^{f} \mu_i(T.C_k) = \nu_j(T2.C'_k)))), \mu_i) | i \in 1, \ldots, n|\!\}$$

2. Para toda vista $V =$ create view V(E_1, ..., E_n) as Q,

$$\theta(V, d) = \theta(Q, d)\{V.E_1 \mapsto Q.A_1, \ldots, V.E_n \mapsto A_n\}$$

3. Si Q es una consulta básica de la forma:

select $e_1, \ldots, e_n$ from R_1 $B_1, \ldots, R_m$ B_m where C_w;

Entonces $\theta(Q, d)$ es igual a:

$$\{\!|(\psi_1 \wedge \ldots \wedge \psi_m \wedge \varphi(C_w\mu, d),\ s_Q(\mu)) \ | \\ (\psi_1, \nu_1) \in \theta(R_1, d), \ldots, (\psi_m, \nu_m) \in \theta(R_m, d), \mu = {\nu_1}^{B_1} \odot \cdots \odot {\nu_m}^{B_m}|\!\}$$

donde

- $\odot$ representa el operador de unión de filas.
- $s_Q(\mu) = \{Q.A_1 \mapsto (e_1\mu), \ldots, Q.A_n \mapsto (e_n\mu)\}$,
- La fórmula de primer orden $\varphi(C, d)$ se define como:
  - Si $C \equiv$ `false` entonces $\varphi(C, d) = \bot$
  - Si $C \equiv$ `true` entonces $\varphi(C, d) = \top$
  - Si $C \equiv e$, con e una expresión aritmética con constantes entonces: $\varphi(C, d) = e$
  - Si $C \equiv e_1 \diamond e_2$, con $\diamond$ un operador relacional, entonces $\varphi(C, d) = (\varphi(e_1, d) \diamond \varphi(e_2, d))$.
  - Si $C \equiv C_1$ `and` C_2 entonces $\varphi(C, d) = \varphi(C_1, d) \wedge \varphi(C_2, d)$
  - Si $C \equiv C_1$ `or` C_2 entonces $\varphi(C, d) = \varphi(C_1, d) \vee \varphi(C_2, d)$
  - Si $C \equiv$ `not` C_1 entonces $\varphi(C, d) = \neg\varphi(C, d)$
  - Si $C \equiv$ `exists` Q supongamos que $\theta(Q, d) = \{\!|(\psi_1, \mu_1), \ldots (\psi_p, \mu_p)|\!\}$. Entonces $\varphi(C, d) = (\vee_{j=1}^{p} \psi_j)$.

4. Si Q es una consulta con operaciones de conjuntos:

   - $\theta(V_1$ union $V_2, d) = \theta(V_1, d) \ \cup \ \theta(V_2, d)$ siendo $\cup$ la unión de multiconjuntos.
   - $(\psi, \mu) \in \theta(V_1$ intersection $V_2, d)$ con cardinal k si y solo si $(\psi_1, \mu) \in \theta(V_1, d)$ con cardinal k_1, $(\psi_2, \mu) \in \theta(V_2, d)$ con cardinal k_2, $k = min(k_1, k_2)$ y $\psi = \psi_1 \wedge \psi_2$.

Obsérvese que si una consulta tiene una sección select de la forma select e_1 E_1, ..., e_n E_n, la notación $s_Q(x)$ refiere al resultado de reemplazar en x cada E_i por su valor e_i.

Estas directrices se aplican para transformar la estructura generada en la fase de análisis sintáctico a partir la definición de la base de datos en las restricciones asociadas a sus relaciones para obtener un caso de prueba.

Representación de fórmulas

Las fórmulas se generan en una estructura de clases propia de la herramienta (figura 2.3). Esto nos permite tratar las fórmulas de forma independiente a cual será el lenguaje final utilizado para resolver las restricciones que representa.

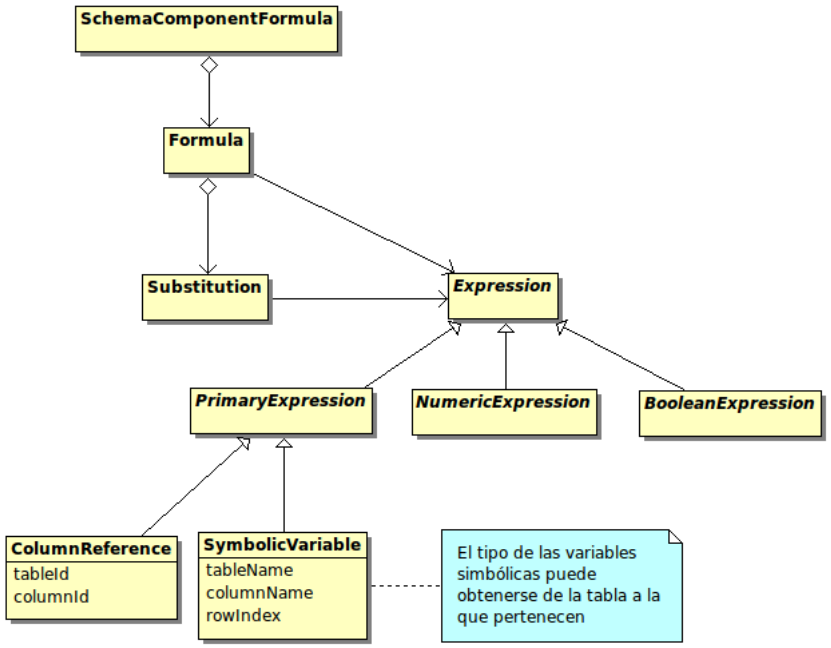

Figura 2.3: Diagrama UML de la estructura utilizada para representar las fórmulas.

Su estructura es análoga a la presentada en la sección anterior. Así las fórmulas asociadas a una determinada relación son una colección de pares de expresiones y sustituciones de los atributos de estas relaciones por expresiones.

En estas fórmulas intervienen variables simbólicas, constantes y expresiones entre las mismas. Las variables simbólicas están tipadas y están identificadas por la tabla que las contiene, el atributo al que dan valor y la fila de la instancia simbólica a la que pertenecen.

En la siguiente sección se indica como se traducen estas fórmulas a lenguajes de programación con restricciones específicos.

2.3.3. Traducción a resolutor específico

La herramienta ha sido diseñada para poder ser utilizada con distintos lenguajes de resolución de restricciones. En un principio se decidió traducir las fórmulas generadas según el método expuesto en la sección anterior a Gecode

[8]. Esta decisión se cambió posteriormente por la generación de código MiniZinc [6], esto fue posible gracias al diseño independiente del lenguaje de restricciones que utilizamos.

Para traducir la representación abstracta de las restricciones a código de un lenguaje programación con restricciones o resolutor concreto hay que realizar las siguientes transformaciones:

- Declarar las variables simbólicas. Cada lenguaje de programación con restricciones tiene su forma de declarar sus variables y el dominio de las mismas. Para hacerlo utilizamos el tipo de cada variable simbólica utilizada y le asignamos un identificador único (formado por la tabla y el atributo al que pertenece y un índice)

- Expresar las restricciones. Los lenguajes de programación con restricciones difieren en la forma de expresar las restricciones. Es necesario transformar las expresiones sobre variables en sus equivalentes en el lenguaje específico. A demás, la herramienta debe traducir el conjunto de fórmulas de cada relación en una disyunción de las mismas.

- Ejecutar el resolutor. Una vez definido el modelo del problema a través de los pasos anteriores, es necesario ejecutar el resolutor. La forma de ejecutarlo y las posibilidades de configuración varían entre los posibles lenguajes y resolutores.

- Establecer formato de salida. Hay que proporcionar al usuario una salida significativa y completa. Para ello hay que especificar el formato de salida del resultado.

Capítulo 3

Desarrollo e implementación

En este capítulo realizaremos una visión general de cómo se ha realizado el desarrollo e implementación del proyecto. En primer lugar nos centraremos en el proceso de desarrollo, analizando en detalle cada fase llevada a cabo y exponiendo las dificultades encontradas que nos ha supuesto enfrentarnos a un proyecto de carácter científico. Posteriormente profundizaremos en la implementación de nuestra herramienta, mediante la ayuda de diagramas UML que clarifiquen lo explicado. Finalmente expondremos un ejemplo de ejecución de nuestra herramienta, explicando cada fase de la ejecución.

3.1. Proceso de Desarrollo

En esta sección describiremos como ha sido el proceso de desarrollo que hemos seguido. Para ello, explicaremos las fases llevadas a cabo, exponiendo los problemas encontrados, las medidas para solventarlos, y los objetivos planteados y logrados para cada una ellas.

3.1.1. Investigación y aprendizaje

Debido a la naturaleza científica de este trabajo, inicialmente fue necesario realizar un proceso de investigación y de aprendizaje exhaustivo. Como se ha explicado anteriormente, nuestro trabajo es una extensión de [3] (ver [5]), en cual se propone la reducción del problema de generación de casos de prueba a un problema de resolución de restricciones. Así, nos encontramos con dos nuevos frentes sobre los que realizar una tarea de investigación: casos de prueba y programación basada en restricciones.

En cuanto al tema de generación de casos de prueba, es una área con la que habíamos tenido contacto previo, gracias a que en asignaturas de cursos anteriores se propuso la implementación de estos, para después ejecutarlos sobre

las prácticas de laboratorio realizadas. Sin embargo, este conocimiento sobre lo que es un caso de prueba era bastante limitado, y era necesario realizar un estudio intenso para tener un conocimiento más profundo y específico de la problemática estudiada.

Por el contrario, la programación basada en restricciones [4] era un área totalmente inexplorada para nosotros. Esto supuso un gran reto, pues debimos rescatar lo estudiado en asignaturas de primer curso, como Matemática Discreta y Lógica Matemática, así como nociones de algoritmia de Metodología y Tecnología de la Programación para tener un conocimiento claro de qué significan las fórmulas lógicas de primer orden que representan las restricciones y el algoritmo de generación de estas, propuesto en [3] y extendido en [5] para consultas existenciales anidadas. El aprendizaje en estas áreas se vio apoyado por las periódicas reuniones que mantuvimos en esta primera fase del proyecto, en las cuales tratamos de resolver dudas que nos surgían durante el proceso de estudio de la teoría de [3].

Por último, y sin olvidar el objetivo final de este trabajo, los casos de prueba surgen de la resolución de las fórmulas de primer orden que representan las restricciones de una tabla o vista SQL. Para poder resolver estas fórmulas se nos propuso utilizar Gecode, un reputado resolutor de restricciones [8]. Gecode (generic constraint development environment) es una herramienta libre, desarrollada en C++ para el desarrollo de sistemas basados en restricciones, que incluye un resolutor de gran rendimiento a la vez que es modular y extensible. De esta forma, paralelamente a los dos frentes de investigación que ya teníamos abiertos, comenzamos un proceso de aprendizaje sobre Gecode, sus posibles usos, y conexiones con nuestra herramienta.

Pronto nos dimos cuenta que esta fase de investigación llevaría más tiempo del que en un principio habíamos estimado. El motivo fue que la teoría expuesta en [3] y la herramienta Gecode [8], no son en absoluto triviales, lo que hizo que empezara a retrasarse el comienzo de la implementación de la herramienta que exponemos. En esta situación, decidimos realizar una planificación del desarrollo, que a continuación detallamos.

3.1.2. Planificación

Puesto que la fase de investigación supuso un retraso importante en el comienzo de la implementación de la herramienta, decidimos realizar una planificación de nuestro desarrollo. El método de desarrollo que hemos aplicado es un desarrollo incremental. Este tipo de desarrollo nos ha permitido realizar una división del trabajo en diferentes fases, de forma que al final de cada una tuviéramos una versión estable de la herramienta, permitiéndonos así incluir nueva funcionalidad en la fase siguiente. Además, esta metodología de desarrollo nos permitió realizar un seguimiento claro de los objetivos que se iban cumpliendo y cuales no. Esto nos ofrecía mayor flexibilidad en cuanto a poder modificar ligeramente la fase siguiente del desarrollo, retrasando objetivos a fases posteriores que fueran menos críticos.

A continuación, se detallan las fases en que dividimos el proceso de desarrollo, indicando para cada una los objetivos marcados, objetivos logrados y conclusiones.

1. *Fase preliminar: Prototipo básico*

   - Objetivo 1: Desarrollo de un prototipo desechable de la funcionalidad básica de la herramienta.
   - Objetivos cumplidos: 1.

   En esta fase el objetivo principal era el desarrollo de un prototipo desechable que mostrara el funcionamiento básico de la herramienta. Consideramos como funcionalidad básica la generación de fórmulas de restricciones de tablas SQL, sin incluir claves primarias ni claves ajenas, y sólo para atributos de tipo enteros. Para este prototipo no se realiza ninguna tarea de parsing, incluyendo las tablas SQL directamente en el código mediante la implementación de una primera estructura C++ para su representación. La única finalidad de este prototipo era que nos pudiéramos hacer una idea del trabajo que teníamos entre manos, pudiendo de esta forma realizar una planificación más exacta del resto de fases del desarrollo. Disponer de este prototipo antes de las vacaciones de Navidad era crucial para no retrasar más el desarrollo de la herramienta.

2. *Fase 1 de desarrollo: Primera versión*

   - Objetivo 1: Diseño e implementación de una estructura C++ extensible para la representación de tablas y vistas SQL.
   - Objetivo 2: Parsing de tablas SQL con atributos de tipo entero.
   - Objetivo 3: Parsing de definiciones de claves primarias y ajenas de tablas SQL.
   - Objetivo 4: Parsing de vistas con atributos de tipo entero y consultas simples (sin subconsultas).
   - Objetivos cumplidos: 1, 2, 3 y 4.

   Esta fase se caracteriza por el diseño e implementación de una estructura C++ que represente tablas y vistas SQL. Esta representación debería permitir su extensión de forma fácil, pues en fases posteriores del desarrollo se tenía la intención de ampliar el lenguaje SQL aceptado. Además, de forma paralela, se comenzó a desarrollar el analizador SQL. Este analizador será el encargado de generar la estructura C++ de las tablas y vistas que el usuario introdujera a la herramienta mediante código SQL.

3. *Fase 2 de desarrollo: Segunda Versión:*

   - Objetivo 1: Extensión de la estructura en C++ para la inclusión de subconsultas existenciales anidadas.
   - Objetivo 2: Parsing de subconsultas existenciales anidadas.

- Objetivo 3: Diseño de un modelo de representación las fórmulas lógicas de primer orden que representan restricciones.
- Objetivo 4: Diseño e implementación del módulo de generación de fórmulas de restricciones para tablas y vistas SQL.
- Objetivo 5: Generación de fórmulas de restricciones para tablas y vistas SQL, estas últimas con subconsultas existenciales.
- Objetivos cumplidos: 1, 2, 3 y 4. El objetivo 5 se cumplió de forma parcial.

En esta fase cabe destacar la inclusión del módulo para la generación de fórmulas de restricciones para tablas y vistas SQL, y el diseño para la representación de estas. Este diseño debería ser lo suficientemente abstracto como para que pueda ser traducido al lenguaje de restricciones del resolutor que el usuario desee utilizar. Esta traducción simplemente requeriría un módulo extra en nuestra herramienta, de forma que tuviera como entrada las fórmulas generadas y las tradujera. El módulo de generación de fórmulas de restricciones es el más importante del desarrollo, pues es el que justifica la realización de este proyecto. También destaca la extensión del analizador SQL con la capacidad para reconocer subconsultas existenciales, así como la ampliación de la estructura C++ para la representación de estas. Como se ha indicado, el objetivo 5 de esta fase se consiguió de forma parcial, pues esta versión de la herramienta generaba fórmulas para tablas y vistas SQL, pero estas últimas, al contrario que el objetivo 4 indicaba, sin subconsultas existenciales. Esta situación fue debida a que en la teoría expuesta en [3], se trata el problema de la generación de fórmulas para subconsultas existenciales de una forma limitada (la subconsulta debe formar parte de una conjunción). Ante esta situación, se nos platearon dos opciones:

a) Seguir con la teoría expuesta en [3], especificando la limitación en cuanto a subconsultas existenciales.

b) Buscar una forma más general para el tratamiento de subconsultas existenciales.

Elegimos la segunda opción, a pesar del riesgo que suponía no encontrar una solución al problema, y retrasar el desarrollo del proyecto en vano. Sin embargo, la motivación de resolver un problema de estas características, junto con la posibilidad de ofrecer una herramienta que fuera independiente de esa limitación, era muy estimulante.

4. *Fase 3 de desarrollo: Tercera versión*

   - Objetivo 1: Extender la estructura C++ para dar soporte a los operadores SQL de conjuntos `UNION` e `INTERSECT` para consultas SQL.
   - Objetivo 2: Parsing de consultas con operadores SQL de conjuntos `UNION` e `INTERSECT`.

- Objetivo 3: Generación de fórmulas para consultas con operadores SQL de conjuntos `UNION` e `INTERSECT`.
- Objetivo 4: Investigar una solución alternativa a la presentada en [3] para el tratamiento de subconsultas existenciales.
- Objetivos cumplidos: 1, 2, 3 y 4.

En esta fase se continua ampliando el lenguaje SQL aceptado por el analizador SQL y la estructura C++ mediante la inclusión de consultas con los operadores de conjuntos `UNION` e `INTERSECT`, y su correspondiente generación de fórmulas de restricciones que contemplan este tipo de consultas. Sin embargo, el principal objetivo de esta fase era encontrar un solución alternativa a la expuesta en [3] para la generación de fórmulas de restricciones de consultas SQL que incluyan subconsultas existenciales anidadas. El resultado de esta investigación puede consultarse en [5].

5. *Fase 4 de desarrollo: Cuarta versión*

   - Objetivo 1: Generación de fórmulas de restricciones para vistas SQL con subconsultas existenciales anidadas.
   - Objetivo 2: Generación de fórmulas de claves primarias y foráneas de tablas SQL.
   - Objetivo 3: Traducción de nuestro modelo de fórmulas de lógica de primer orden a resolutor específico.
   - Objetivos cumplidos: 1, 2 y 3.

   Con la posibilidad de generar fórmulas de restricciones para subconsultas existenciales anidadas y claves primarias y ajenas de tablas SQL, completamos un gran paso en cuanto al conjunto de lenguaje SQL aceptado y usabilidad de la herramienta, por lo que decidimos empezar a generar los casos de prueba, objetivo final del proyecto. Los casos de prueba resultan de la resolución de las fórmulas de restricciones, que hasta entonces, genera la herramienta. Como se mencionó en la sección de investigación 3.1.1, la resolución de restricciones es un campo ampliamente investigado, lo que permite disponer de resolutores de gran rendimiento de software libre. Inicialmente se propuso que utilizáramos el reputado resolutor de restricciones Gecode [8]. Sin embargo, la curva de aprendizaje de este resolutor resultó ser demasiado pronunciada, pues no es una herramienta trivial, dado su bajo nivel del abstracción, y el tiempo del que disponíamos era reducido. Por lo tanto, se buscaron alternativas, encontrando en MiniZinc [6] la más adecuada.

   MiniZinc es un lenguaje de modelado de restricciones que permite la representación de problemas de restricciones. MiniZinc nos ofrecía una forma más abstracta y cercana a nuestra representación interna que Gecode para la traducción de las fórmulas de restricciones, haciendo que la curva de aprendizaje fuera mucho menos pronunciada en comparación con la de

Gecode. Además, la última distribución de MiniZinc incluye el resolutor de restricciones *G12/CPX*, por lo que podíamos ejecutar directamente los modelos de MiniZinc resultantes de la traducción de la fórmulas de restricciones. Cabe resaltar que MiniZinc, o su versión de más bajo nivel *FlatZinc* (la última distribución de MiniZinc incorpora conversores de MiniZinc a FlatZinc), es un lenguaje de modelado de restricciones aceptado como entrada por un gran número de resolutores, incluido el propio Gecode. Esta característica de MiniZinc y FlatZinc fue muy importante a la hora de tomar la decisión de utilizarlo como lenguaje de restricciones, pues dota a nuestra herramienta de poder ejecutar el modelo generado sobre el resolutor que el usuario considere más adecuado. Algunos resolutores de restricciones que implementan MiniZinc o FlatZinc como lenguaje de restricciones son: G12/CPX, Gecode [8], ECLiPSE, SICtus Prolog, JaCoP, SCIP.

3.2. Implementación

En esta sección detallaremos los aspectos que consideramos mas importantes sobre la implementación de la herramienta que exponemos. En primer lugar, realizaremos un repaso a la arquitectura del sistema y explicaremos los módulos que lo componen. A continuación veremos cómo es la interacción entre los distintos módulos.

3.2.1. Descripción de la arquitectura

El esquema de ejecución de nuestra herramienta tiene una estructura de pipeline (ver sección 2.3). Para su realización, se identificaron los distintos módulos funcionales que conforman la arquitectura del proyecto. Estos módulos están disponibles a través de métodos de acceso, que definen la interfaz del módulo para poder ser utilizado. Este tipo de arquitectura nos permitió dividir y paralelizar de forma clara y simple el trabajo a realizar, de forma que podíamos ir avanzando en diferentes módulos al mismo tiempo, con el ahorro de tiempo que ello suponía. La única dificultad fue definir claramente los métodos de acceso a los módulos, pues los módulos son dependientes unos de otros, y a la hora de ensamblarlos deberíamos no tener dificultades. Un esquema de los módulos que componen nuestro sistema, y las dependencias entre ellos, se puede ver en la figura 3.1.

A continuación realizaremos una descripción de cada módulo, explicando características relevantes de cada uno, y mostrando algunos diagramas UML que ayuden a su comprensión.

Módulo de análisis SQL: SQL_parser

Como ya se ha mencionado, nuestra herramienta incorpora un analizador de lenguaje SQL. El objetivo de este analizador no es comprobar la corrección de

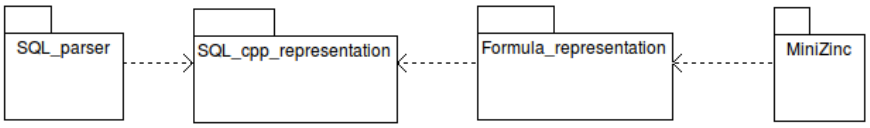

Figura 3.1: Diagrama UML de los módulos que componen el sistema y sus dependencias.

las sentencias introducidas, sino generar una estructura C++ que represente las sentencias SQL dadas a las herramienta, ya se mediante la entrada estándar o mediante un fichero. SQL es un lenguaje con una gramática muy amplia, por lo que hemos limitado la gramática que acepta este analizador a un subconjunto representativo de la misma, descrito en la sección 2.2. No obstante, en futuras versiones de esta herramienta uno de los objetivos marcados es la ampliación de la gramática aceptada. Esta mejora dotaría de mayor libertad a los usuarios a la hora de poder seleccionar las tablas o vistas sobre las que obtener casos de prueba, sin necesidad de reescribir las sentencias de definición de estas.

Este analizador está implementado mediante la utilización de la herramienta GNU Bison [11] y Flex [10], y actualmente acepta código SQL de definición de bases de datos con las características indicadas en la sección 2.2 exceptuando las consultas de agregación. Consideramos importante la implementación de agregados (GROUP BY, HAVING) con la técnica presentada en [3], sin embargo esta característica inicialmente planeada ha quedado fuera de la versión actual del prototipo por restricciones de tiempo.

Módulo para la representación de SQL en C++: SQL_cpp_representation

El objetivo del módulo *SQL parser* es la generación de una estructura que representa las tablas y vistas indicadas por el usuario. Para ello, creamos un módulo que diera soporte para poder realizar esta representación. Este módulo está diseñado con especial cuidado, pues queríamos que fuese extensible de una forma sencilla. Este requerimiento que nos impusimos se debe a que la gramática admitida por el analizador de SQL puede extenderse en versiones futuras, por lo que se deberían definir nuevas estructuras para su representación en caso de que fuera necesario. En la figura 3.2 se puede observar un diagrama de clases que componen este módulo.

Módulo para la representación y generación de fórmulas de restricciones: Formula_representation

Una vez se ha generado una estructura en C++ que represente las tablas y vistas SQL suministradas por el usuario, el objetivo es generar las fórmulas de restricciones sobre la tabla o vista que se indique. Para ello, necesitábamos dar soporte para la representación de las fórmulas de restricciones, de forma que

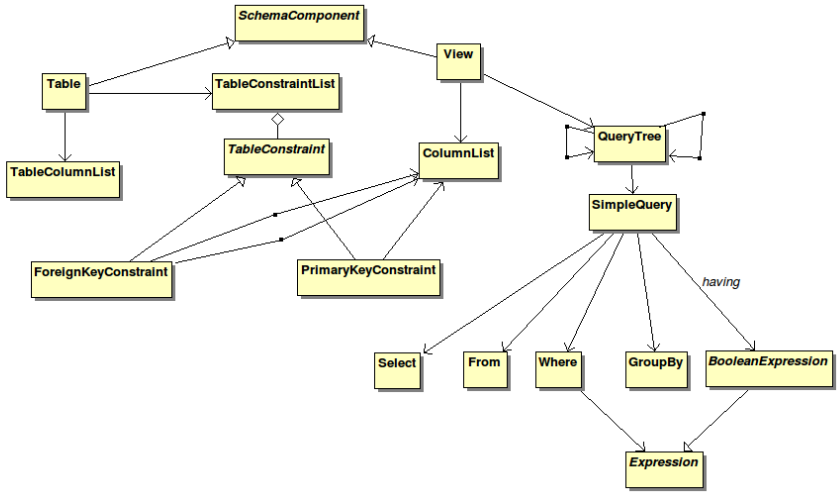

Figura 3.2: Diagrama UML de las clases que componen el módulo para la representación en C++ de tablas, vistas, consultas y expresiones SQL.

fuera lo más fiel a la representación descrita en la sección 2.3.2, y adoptada de [3]. Con este propósito, decidimos crear el módulo *Formula representation*. En la figura 3.3 se puede observar un diagrama UML que representa este módulo.

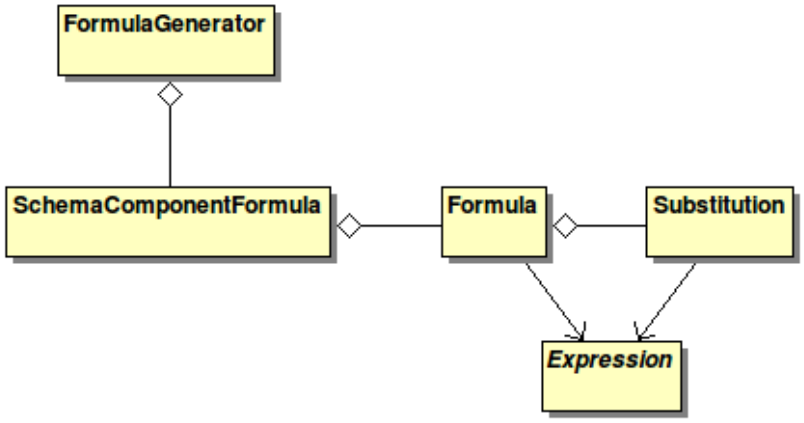

Figura 3.3: Diagrama UML de las clases que componen el módulo para la representación de fórmulas de restricciones.

Cómo se puede observar, éste módulo incluye una clase *FormulaGenerator*. Esta clase es la encargada de generar la fórmula de restricciones para una tabla

o vista indicada por el usuario. Para generar éstas fórmulas, se ha implementado el algoritmo de generación de fórmulas descrito en 2.3.2.

Módulo traductor a lenguaje de restricciones de un resolutor específico

La representación de fórmulas de restricciones que utilizamos es una representación genérica, próxima al lenguaje matemático. Esto nos da la posibilidad de poder traducir esa representación a modelos de restricciones que acepten resolutores de restricciones concretos. Únicamente bastaría con implementar un nuevo módulo que tome como entrada un fórmula de restricciones con la representación propia y traducirla al lenguaje concreto.

El modelo de restricciones, traducido al lenguaje de un resolutor concreto o de un lenguaje de programación de restricciones específico puede ser guardado en un fichero. De esta forma, podríamos ejecutar el resolutor en cuestión, indicando que tome como entrada el fichero guardado. El resolutor asignará valores a las variables simbólicas que componen el modelo obteniendo, si existiera, el caso de prueba deseado.

Actualmente, nuestra herramienta realiza la traducción al lenguaje de modelado de restricciones MiniZinc [6]. Elegimos este lenguaje por ser aceptado por un gran número de resolutores de restricciones. Hemos tenido especial cuidado a la hora de mostrar el resultado proporcionado por el resolutor, pues considerábamos útil el hecho de generar sentencias SQL que poblaran las tablas de la base de datos involucradas en la vista o tabla de la que se buscaba un caso de prueba. Esta sentencias están introducidas mediante:

```
INSERT INTO table_name (atributos) VALUES (valores)
```

Esto permitiría ejecutar estas sentencias SQL en el gestor de bases de datos que este utilizando el usuario, y poder así ejecutar los casos de prueba.

3.2.2. Interacción entre módulos

A continuación explicaremos distintas secuencias de ejecución que puede tener nuestra herramienta. Para facilitar la comprensión nos ayudaremos de diagramas de secuencias.

En la figura 3.4 se describe el comportamiento de nuestro sistema cuando la entrada del sistema es un fichero SQL con sentencias de definición de tablas y vistas (ver apéndice B.1 para ver un ejemplo de fichero de entrada). En este caso, el módulo de análisis de SQL leerá el fichero indicado y creará una estructura C++ que represente a las tablas y vistas que estén definidas. Al final de este proceso, se preguntará al usuario por una de las tablas o vistas para la que se quiera generar un caso de prueba, y se comunicará al módulo de generación de fórmulas que genere una fórmula de restricciones para la tabla o vista indicada. Este módulo preguntará al usuario por el número máximo de filas que pueden tener las tablas involucradas en la generación de la fórmula de restricciones

para la tabla o vista que se indique. Una vez que haya generado esta fórmula, se traducirá a un modelo en lenguaje MiniZinc, el cual podrá ser ejecutado posteriormente por un resolutor que acepte este lenguaje para obtener un caso de prueba.

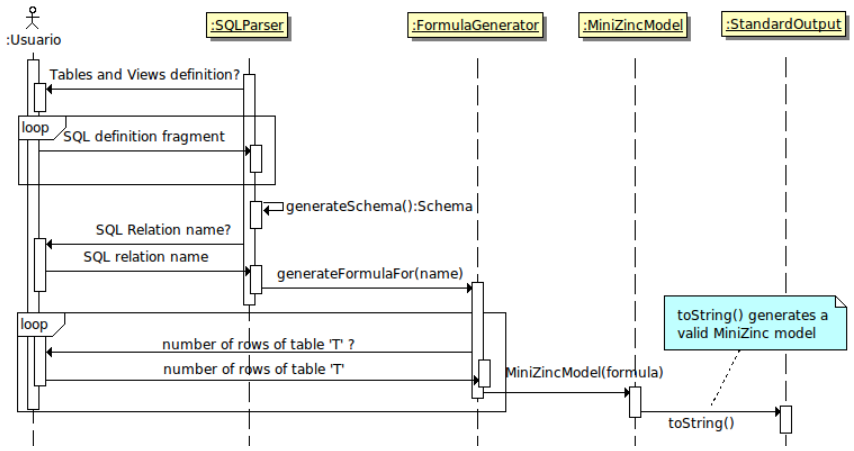

Figura 3.4: Diagrama de secuencias con fichero de entrada.

En la figura 3.5 se muestra una fase inicial de interacción con el usuario. En el caso de que el usuario no indique un fichero de entrada en el que se encuentren las definiciones de tablas y vistas, se pueden introducir en la herramienta de forma interactiva. Una vez que el usuario ha introducido todas las sentencias SQL de definición de tablas y vistas, se procederá al análisis de estas, continuando con el mismo proceso que en el caso anterior.

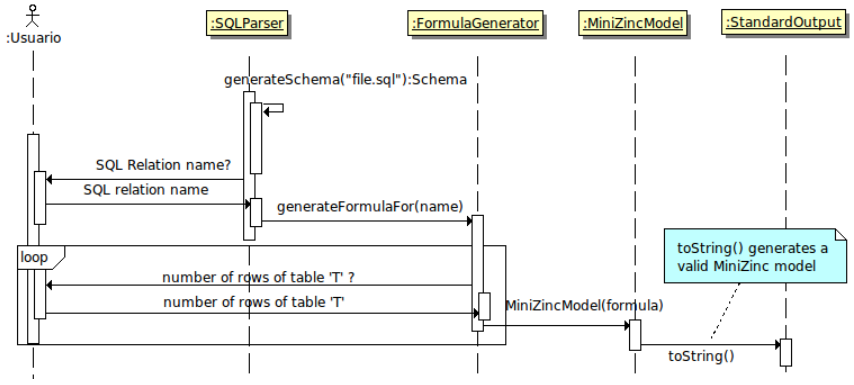

Figura 3.5: Diagrama de secuencias interactivo.

La salida que proporciona nuestra herramienta constituye un modelo MiniZ-

inc válido. Este modelo se puede guardar en un fichero, indicando su ruta con la opción `--output ruta` (ver manual en apéndice C) o redirigiendo la salida estándar, para posteriormente obtener sus soluciones a través del resolutor de restricciones.

3.3. Ejemplo de ejecución

En esta sección presentamos un ejemplo de ejecución de la herramienta, con el objetivo de ayudar a entender el funcionamiento de los componentes del sistema antes explicados. Dada la estructura en pipeline de la herramienta, mostraremos y comentaremos los resultados de cada fase de la ejecución.

El ejemplo propuesto consiste en un juego de mesa, compuesto por un tablero, unos jugadores y sobre el tablero se tienen fichas de los jugadores. El tablero de juego tiene forma rectangular, y está compuesto por casillas, indexadas por su número de columna y número de fila (coordenadas `x` e `y` respectivamente). En cada casilla del tablero sólo puede haber una ficha de un jugador, pero un jugador puede tener varias fichas sobre el tablero. Supongamos que el juego se gestiona mediante una base de datos, en la que se tienen las siguientes tablas para almacenar los jugadores y el estado del tablero:

```
create table player {  id int primary key  };

create table board {
  int x,
  int y,
  int id,
  primary key x,y;
  foreign key(id) references player(id);
  };
```

dónde `x,y` son las coordenadas de la pieza del jugador identificado mediante `id` en el tablero.

Una de las situaciones del juego consiste en eliminar a los jugadores tengan todas sus fichas amenazadas. Una ficha está amenazada si tiene la misma coordenada `x` o la misma coordenada `y` que alguna ficha de otro jugador. Las dos vistas siguientes seleccionan los jugadores que cumplen con estas premisas:

```
create view nowPlaying(id) as
  select p.id
  from player p
  where exists (select b.id from board b where b.id=p.id);

create view checked(id) as
  select p.id
  from player p
  where exists (select n.id from nowPlaying n where n.id = p.id) and
        not exists (select b1.id from board b1
```

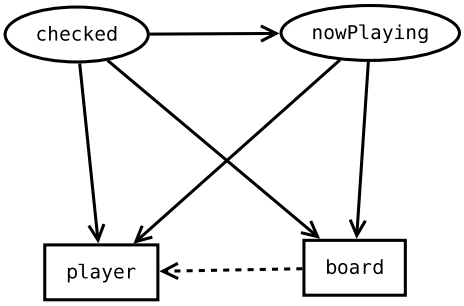

Figura 3.6: Grafo de dependencias entre las tablas `player` y `board` y las vistas `nowPlaying` y `checked` del ejemplo propuesto.

```
where b1.id = p.id and
      not exists
     (select b2.id from board b2
      where (b2.x - b1.x) * (b2.y-b1.y)=0 and
            (b1.id <> b2.id)));
```

Supongamos que el usuario ejecuta nuestra herramienta con un fichero de entrada `input.sql` con las anteriores sentencias SQL (detallado en el apéndice B.1). Este fichero será analizado por el analizador SQL, generando como resultado una estructura C++ que representa las definiciones de tablas y vistas SQL que contiene. Por simplicidad, supongamos que el usuario indica a la herramienta que la tabla `player` y la tabla `board` tienen dos filas como máximo, y que el rango de los atributos de ambas tablas es de $[1, 5]$. La representación C++ inherente a las sentencias SQL que contiene el fichero `input.sql` puede consultarse en el apéndice B.2.

Una vez que el analizador SQL ha generado la anterior representación C++, el módulo generador de fórmulas tomará como entrada esta estructura y el nombre de la tabla o vista SQL de la cual se quiere generar un caso de prueba. En nuestro caso, generaremos un caso de prueba para la vista `checked`, por ser la de mayor interés.

La generación de fórmulas de restricciones para una vista, viene determinada por la tablas y vistas de las que depende, es decir, las que forman la cláusula `from`, además de la expresión de la cláusula `where`. Estas dependencias se pueden representar mediante un grafo, tal y como se puede ver en la figura 3.6.

A continuación ilustraremos cómo generaría nuestra herramienta la fórmula de la vista `checked`. Por simplicidad, explicaremos cómo se genera la fórmula correspondiente para cada uno de sus componentes, de forma que en el último paso se compondrían en una única fórmula. Señalar también que las fórmulas expuestas no están completas, pues se corresponden a fragmentos interesantes de las misma. Estos fragmentos pertenecen a la primera fila de cada fórmula, por ser el resto de filas duales. Se puede consultar la fórmula entera en el apéndice B.3. La notación para representar las filas ha sido adoptada de [3], en la cual

una fila es una substitución de la forma:

$$\{\texttt{x} \rightarrow \texttt{expr}\}$$

Esta notación indica que la variable simbólica `x` se sustituye por la expresión `expr`.

La primera fórmula que procedemos a explicar se corresponde con la tabla `player` (ver B.3.1). La primera fila de la tabla:

$$\{\texttt{player.id} \rightarrow \texttt{player.id.0}\}$$

representa la sustitución de la variable simbólica `player.id` por el valor que tiene el campo `id` de la fila número 0 en la tabla `player`. En la fórmula se indica que dos jugadores no pueden tener un mismo `id`, es decir, no puede haber dos filas de la tabla con el mismo valor en el campo `id`. Para la primera fila de la tabla, se tiene:

$$\texttt{player.id.0 != player.id.1}$$

Esto es así por que, en la definición SQL, se indica que el atributo `id` de la tabla `player` es clave primaria:

```
#(player)={(player.id.0 != player.id.1), {player.id -> player.id.0}), ...}
```

Veamos ahora como sería la fórmula para la tabla `board` (ver B.3.2). La primera fila de la tabla:

$$\{\texttt{board.x} \rightarrow \texttt{board.x.0, board.y} \rightarrow \texttt{board.y.0, board.id} \rightarrow \texttt{board.id.0}\}$$

indica que las variables simbólicas `board.x`, `board.y` y `board.id` se sustituyen por los valores de los campos `x`, `y` e `id` de la fila número 0 de la tabla. Atendiendo a la definición de la tabla `board`, encontramos dos tipos de restricciones. La primera restricción es de clave primaria para los atributos `x e y` , que indica que no puede haber dos piezas en la misma casilla del tablero, es decir, que tengan la misma coordenada `(x,y)`. Para la primera fila:

$$\texttt{((board.x.0 != board.x.1)} \vee \texttt{(board.y.0 != board.y.1))}$$

Por otro lado, tenemos una restricción de clave foránea para el atributo `id`. Este atributo hace referencia al atributo `id` de la tabla `player`, por lo tanto, tiene que ser igual a alguno de los valores de `id` de la filas de la tabla `player`. De esta forma, para la primera fila de la tabla se tiene:

$$\texttt{((board.id.0 == player.id.0)} \vee \texttt{(board.id.0 == player.id.1))} \wedge \texttt{((board.id.1 == player.id.0)} \vee \texttt{(board.id.1 == player.id.1))}$$

Así, la fórmula tendría la siguiente forma:

```
#(board)={
(((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1))),
{board.x -> board.x.0,board.y -> board.y.0,board.id -> board.id.0}), ...}
```

La siguiente fórmula de restricciones que mostramos corresponde a la vista auxiliar `nowPlaying`. La primera fila que generaría esta vista (ver B.3.3 para la fórmula completa):

$$\{\texttt{nowPlaying.id} \rightarrow \texttt{player.id.0}\ \}$$

nos indica que el atributo `id` será sustituido por el valor del campo `id` de la fila número 0 de la tabla `player`. Esta vista indica los `id`s de los jugadores que están jugando actualmente. Como se puede observar en su definición o el grafo de dependencias (figura 3.6), esta vista depende de la tabla `player`, y contiene una subconsulta existencial (clausula `where`), que depende tanto de la tabla `player` externa como de la tabla `board`. La idea subyacente de esta vista es seleccionar los `id` de jugadores que existan en el conjunto de jugadores con fichas en el tablero, es decir, para la primera fila se tiene:

$$\texttt{(board.id.0 == player.id.0)} \vee \texttt{(board.id.1 == player.id.0)}.$$

Así, lo fórmula para la vista `nowPlaying`, tendría la forma:

```
#(nowPlaying)={
(((...) AND (( ... AND (board.id.0 == player.id.0)) OR (...  AND
(board.id.1 == player.id.0)))),{nowPlaying.id -> player.id.0}), ...}
```

La fórmula que presentamos en último lugar se corresponde con la vista `checked` (ver apéndice B.3.4), para la cual buscaremos un caso de prueba positivo. Cómo se puede apreciar en su definición, es una vista bastante compleja, con varias subconsultas anidadas, y que depende, no solo de las tablas `player` y `board`, si no también de la vista auxiliar `nowPlaying`. La primera fila que generaría la ejecución de esta vista sería:

$$\{\texttt{checked.id} \rightarrow \texttt{player.id.0}\}$$

que asignaría a la variable simbólica `checked.id` el valor del campo `id` de la primera fila de la tabla `player`. El objetivo de esta vista es decidir si dos fichas, de jugadores distintos, tienen la misma coordenada `x` o coordenada `y` del tablero de juego. Esta propiedad se describe mediante la siguiente expresión:

$$\texttt{((board.x.0 - board.x.1)*(board.y.0 - board.y.1) == 0)} \wedge \texttt{(board.id.1 != board.id.0)}.$$

Así, la fórmula tendría la siguiente forma:

```
#(checked)= {
( ... AND ...
(not (((((board.x.0 - board.x.1) * (board.y.0 - board.y.1)) == 0)
AND (board.id.1 != board.id.0)) OR ...)),
{checked.id -> player.id.0}), ...}
```

En realidad, nuestra herramienta genera una única fórmula de restricciones para la tabla o vista que se seleccione. Como se mencionó anteriormente, si una vista depende de otras tablas o vistas, se calcularán las fórmulas de restricciones asociadas a estas, ya que formarán parte de la fórmula de restricciones asociada a la vista original, junto con la expresión que contenga la expresión de la cláusula `where`. En este caso, la fórmula para la vista `checked`, en una representación abstracta, podría ser:

$$\theta(checked) = \theta(nowPlaying) \wedge \theta(player) \wedge \theta(board) \wedge where_{checked}$$

Continuando con nuestro ejemplo, procedemos a buscar un caso de prueba positivo para la vista `checked`. Por lo tanto, tras generar su correspondiente fórmula de restricciones, se inicia la etapa de traducción de dicha formula al lenguaje de restricciones del resolutor específico. Para ello, elegimos MiniZinc como lenguaje de modelado de restricciones, pues es un lenguaje aceptado por un gran número de resolutores de restricciones. La última distribución incluye el resolutor G12, por lo que usaremos dicho resolutor por comodidad. De esta forma, se obtendría un fichero de salida, supongamos *output.mzn* (ver B.4). Este fichero contiene el modelo de restricciones para la vista `checked` en lenguaje MiniZinc, y podrá ejecutarse posteriormente, obteniendo, si existe, el caso de prueba positivo para dicha vista. En nuestro caso, y con el objetivo de obtener resultados significativos, hemos reducido el dominio de las variables simbólicas que componen el modelo MiniZinc, pudiendo éstas tomar valores entre 1 y 5. Tras ejecutar este fichero se obtiene el siguiente resultado:

```
INSERT INTO board(id,x,y) VALUES (1,1,1);
INSERT INTO board(id,x,y) VALUES (2,1,5);
INSERT INTO player(id) VALUES(1);
INSERT INTO player(id) VALUES(2);
```

Como se puede ver, el resultado está compuesto por sentencias SQL. Estas sentencias, ejecutadas en un entorno SQL, poblarán las tablas involucradas en la vista sobre la que se quiere obtener un caso de prueba. Cabe señalar que este resultado puede variar, dependiendo del resolutor usado o del método de búsqueda de la solución empleado, pero si existe un caso de prueba positivo, el modelo de restricciones que usamos asegura que se encuentra. Distintos casos de prueba pueden obtenerse pidiendo al resolutor que porporcione más soluciones.

Tras ejecutar las sentencias obtenidas del resolutor como resultado, la instancia de base datos que define el caso de prueba tendría la siguiente forma:

player
id
1
2

board		
id	x	y
1	1	1
2	1	5

Con esta instancia de bases de datos, la configuración del tablero de juego tendría la configuración siguiente:

2				
1				

Dónde las casillas en blanco indican que no hay fichas en ellas. Además, el jugador con `id = 2` tiene un ficha en la casilla situada en la fila 5 y columna 1 del tablero (coordenadas `(y,x)` respectivamente), mientras que el jugador con `id = 1` tiene una ficha situada en la casilla de la fila 1 y columna 1 (coordenadas `y,x` respectivamente).

Si ahora realizamos la siguiente consulta a nuestro gestor de bases de datos:

```
select * from checked;
```

El resultado que se obtendría sería:

id
1
2

A la vista de la configuración del tablero, es el resultado esperado, pues el jugador con `id = 1` y el jugador con `id = 2` poseen una ficha cada uno, que además están mutuamente amenazadas.

Capítulo 4

Conclusiones y trabajo futuro

4.1. Trabajo realizado

En el proyecto se han implementado las ideas propuestas en [3] para la obtención de casos de prueba para bases de datos SQL y se han extendido las mismas para dar solución a consultas existenciales anidadas [5]. El desarrollo se ha concretado en tres módulos principales: parser, generador de fórmulas y traductor a lenguajes específicos o resolutores, cuyas principales características mostramos a continuación.

- *Parser*

  Hemos desarrollado un analizador sintáctico de un subconjunto de SQL (ver sección 2.2 y apéndice A). Esto nos permite obtener la estructura de la base de datos que deseamos estudiar a partir de su definición en lenguaje SQL.

- *Generador de fórmulas*

  A partir de la representación de la base de datos obtenida por el parser y aplicando las técnicas propuestas en [3] y [5] generamos las restricciones que deben cumplir los casos de prueba. Estas restricciones se generan en un formato interno, que nos permite posteriormente traducirlas a diversos lenguajes de programación con restricciones.

- *Generador de código MiniZinc*

  Nuestra herramienta traduce las fórmulas generadas en el módulo anterior a código MiniZinc [6]. El código generado (ver apéndice B.4) se divide en la declaración de las variables, la definición de las restricciones, la llamada al resolutor y el formato de salida. Este código será posteriormente ejecutado en un resolutor para encontrar, si existiera, un caso de prueba.

Actualmente la salida del resolutor tiene un formato compatible con SQL (ver resultado en la página 41), de forma que se puede introducir fácilmente el caso de prueba generado en la base de datos.

4.2. Trabajo futuro

A partir del trabajo realizado se han identificado una serie de necesidades y posibles mejoras de la herramienta. Se presenta a continuación una clasificación basada en la prioridad de las mismas.

- *Agregados*

  El uso de agregados en bases de datos está ampliamente extendido (sentencias `GROUP BY`, `HAVING`). A diferencia de otras sentencias SQL que pueden ser traducidas a otras equivalentes que nuestra herramienta ya soporta (apéndice D), los agregados necesitan un trato diferenciado. El uso de agregados está contemplado en [3], y la inclusión en nuestra herramienta no debería suponer grandes complicaciones. Sin embargo, y aún considerando necesaria esta característica para poder considerar completa la herramienta, la versión actual del prototipo no admite todavía este tipo de sentencias.

- *Tipos de datos*:

  Un gran número de bases de datos utiliza sólo datos de tipo entero o cadenas de caracteres. Actualmente la herramienta da soluciones sólo para tipos enteros, sin embargo está diseñada para trabajar con diferentes tipos de datos. Una de las mayores dificultades a la hora de admitir nuevos tipos de datos es la traducción a restricciones de sus funciones y condiciones específicas (por ejemplo los filtros mediante expresiones regulares o las concatenaciones en cadena de caracteres) y mantener la sintaxis de las condiciones más generales que afectan a varios tipos de datos (comparaciones, desigualdades, etc.). El trabajo de modelar como restricciones las condiciones sobre cadenas de caracteres está en gran parte implementado en un trabajo reciente [12] por lo que la inclusión de esta característica no sería muy costosa.

- *Configuración del dominio de las variables*

  El prototipo genera fórmulas sobre variables enteras sin restringir su dominio. Este comportamiento es el esperado, ya que no se puede conocer a priori un dominio para los valores enteros que sea válido para todas las bases de datos. Sin embargo, esto puede hacer que el tiempo usado para encontrar soluciones sea considerablemente más elevado que si el dominio de las variables fuera restringido. Parece razonable entonces ofrecer al usuario la configuración del dominio de las variables, acelerando el proceso de búsqueda de soluciones y dejando en manos del usuario la decisión de los valores válidos para su base de datos.

- *Valores nulos*

  Los valores nulos (NULL) sirven para representar que un determinado dato no existe en la base de datos. En [13] se muestra el enfoque a adoptar para tratar con valores nulos en white-box testing para SQL. Se debe usar la lógica ternaria (lógica con tres valores: Cierto, Falso y Desconocido) que utiliza SQL para tratar con datos nulos. Además, sería necesaria la introducción de una representación adecuada de los valores nulos en el modelo de restricciones. El uso de valores nulos es interesante, sin embargo su implementación no es tan simple como a priori pudiera parecer.

- *Integración con otras herramientas*

  La herramienta sería de gran utilidad si pudiera integrarse con otras herramientas complementarias de depuración y prueba. Por ejemplo, en la herramienta presentada en [2] que puede ejecutarse online en la siguiente dirección: http://in2test.lsi.uniovi.es/sqlfpc/SQLFpcWeb.aspx se subdivide una consulta en un conjunto de consultas que conforman un recubrimiento exhaustivo. La resolución de dichas consultas con nuestra herramienta supondría la obtención del conjunto de prueba para la consulta en cuestión.

- *Ampliación de la sintaxis admitida*

  Actualmente el prototipo admite un subconjunto reducido del lenguaje SQL (ver sección 2.2 y apéndice A). Ampliar el lenguaje admitido supondría una clara mejora de la herramienta, que podría utilizarse sin necesidad de reescritura en un mayor número de bases de datos. Es conocida la equivalencia entre algunas sentencias SQL y las sentencias admitidas por la herramienta (ver apéndice D). Así, podemos afrontar este objetivo desde dos perspectivas diferentes: transformar mediante un análisis previo estas sentencias en sentencias ya soportadas por la herramienta o incluirlas directamente en el parser del que dispone actualmente el prototipo. Algunas de las sentencias SQL que sería interesante que nuestra herramienta admitiera son:

  - **Subconsultas de tipo `ANY`, `IN`, `ALL`**. A través de la conversión en subconsultas existenciales equivalentes.
  - **Subconsultas en la clausula `FROM`**. Esto se puede conseguir mediante la transformación a vistas intermedias de dichas subconsultas.

- *Optimización de fórmulas generadas*

  En la versión actual del prototipo, las fórmulas generadas no pasan por un proceso de optimización que las transforme en fórmulas más claras o más eficientes. Las fórmulas que actualmente generamos tienen muchas expresiones que podrían simplificarse como por ejemplo `True OR X` se simplifica a `X`, la transformación a fórmulas más simples facilitaría la comprensión

del las mismas. Además, a través del estudio estadístico del tiempo de ejecución de formas equivalentes de expresar las restricciones podría guiarnos a una generación más eficiente de las mismas.

- *Expresividad de salida del parser*

  Si bien el analizador sintáctico de la herramienta no pretende ser un compilador de SQL, sí debería dar una salida significativa de los motivos por los cuales rechaza cierta entrada, sobre todo si la misma está bien formada pero no está soportada actualmente por la versión actual del prototipo.

- *Ordenación de resultados de salida*

  Se ha observado que, en ocasiones, la salida de texto que representa el caso de prueba obtenido por el prototipo, al introducirse directamente en la base de datos como código SQL, no instancia la base de datos con el caso de prueba completo. Esto es debido a las restricciones de las claves ajenas (FOREIGN KEY) y se puede solucionar cambiando el orden de introducción de las sentencias en la base de datos. Esto no debe entenderse como un error ya que la salida debe considerarse como una instancia de una base de datos independiente del orden de las mismas. La presentación como sentencias SQL se proporciona para dar claridad y facilitar la introducción de la salida en la base de datos. La resolución de este problema no es trivial para algunos casos (ciclos en las relaciones de clave ajena, por ejemplo), por lo que debe estudiarse si la solución debe venir de la herramienta o de la interacción del usuario con la base de datos.

- *Traducción a otros lenguajes de restricciones y resolutores*

  El prototipo permite la traducción de las restricciones generadas a diferentes lenguajes de restricciones (ver sección 2.3.3). Actualmente solo se traduce a código MiniZinc [6], pero sería interesante ampliar la variedad de lenguajes a las que la herramienta traduce las restricciones.

4.3. Aplicaciones

En esta sección se exponen las utilidades prácticas de la herramienta y las posibles aplicaciones futuras del proyecto u otras herramientas que implementen estas ideas.

Dentro de las aplicaciones inmediatas, el prototipo puede usarse directamente para la obtención de casos de prueba para conjuntos de vistas definidas sobre bases de datos SQL. Además sirve para el estudio de la implementación de las ideas desarrolladas y puede usarse para estudiar la complejidad y eficiencia de las mismas, buscar optimizaciones en la generación de fórmulas y conocer sus ventajas y limitaciones.

La integración con otras herramientas de depuración [1] y prueba [2], así como la implementación de las mejoras propuestas en la sección anterior puede ampliar significativamente la utilidad práctica de la herramienta.

En el entorno del software de generación de conjuntos de prueba para SQL como [2] la herramienta podría ser utilizada directamente para la obtención de casos de prueba concretos. Esta parece una de las utilidades de la herramienta que podría cumplir a corto plazo.

También se podría utilizar la herramienta para poblar bases de datos de forma automática, generando casos de prueba para consultas que requieren un gran número de instancias.

Otra posibilidad es usar la herramienta como complemento para depuradores de SQL como [1], generando como casos de prueba alguna de las instancias que estas herramientas necesitan.

4.4. Conclusiones

La generación de casos de prueba para SQL es un tema de investigación actual. En este contexto, nuestro proyecto desarrolla ideas innovadoras presentadas en [3] e incluso hace sus propias aportaciones al campo. Las ideas sobre cómo transformar a restricciones consultas existenciales anidadas se han concebido en el seno de nuestro proyecto y la publicación de las mismas está actualmente en proceso [5].

El proyecto ha consistido en la implementación de un prototipo para la obtención de casos de prueba positivos de consultas SQL. Consideramos un caso de prueba positivo a una instancia de la base de datos tal que la relación estudiada devuelve una solución no vacía. Para ello, se realiza un análisis sintáctico del código SQL que define la base de datos que queremos estudiar y se generan las restricciones que debe cumplir una instancia para ser un caso de prueba positivo. Estas restricciones se traducen posteriormente a un lenguaje concreto de programación con restricciones, en nuestro caso MiniZinc. Finalmente un resolutor encuentra, si existiera, la instancia buscada, que se presentará al usuario en forma de inserciones en la base de datos.

La herramienta desarrollada constituye un prototipo novedoso en el campo de las bases de datos que podrá ser de verdadera utilidad práctica una vez actualizada con las ideas presentadas en la sección de trabajo futuro (sección 4.2). Actualmente existe una carencia de herramientas complementarias de depuración y prueba para el desarrollo de bases de datos relacionales. Trabajos como el presente muestran que este tipo de herramientas es factible y que presumiblemente supondrán una mejora en el desarrollo de bases de datos.

Apéndice A

Gramática

En esta sección se muestra la estructura de la gramática actualmente soportada por el proyecto.

Es una gramática escrita para GNU Bison [11] en la que por motivos de espacio y para facilitar la comprensión se han eliminado los fragmentos de código relacionados con cada rama de *parsing* así como la definición de tipos de cada estructura. El código completo del parser y del analizador léxico puede ser consultado en los archivos SQL.y y SQL.l en la siguiente carpeta del repositorio del código: http://stcg.googlecode.com/svn/trunk/src/SQL_analysis/SQL_parser/ .

La gramática, expresada en la sintaxis parecida a BNF que utilizan Yacc y Bison [11] y basada en la gramática propuesta en [9] es la siguiente:

Notese que los *tokens* de la gramática están escritos en mayúsculas.

```
%start start

start:
/*empty*/
table_definition
|start SC table_definition
|view_definition
|start SC
;

view_definition:
create_view table_name vcl AS query_expression
;

create_view:
CREATE VIEW
;

vcl:
```

```
/*empty*/
|LP view_column_list RP
;

view_column_list:
column_name_list
;

query_expression:
non_join_query_expression
;

non_join_query_expression:
non_join_query_term
|query_expression UNIONOP query_term /*UNION*/
|query_expression INTERSECTOP query_term /*INTERSECTION*/
;

non_join_query_term:
non_join_query_primary
;

query_term:
non_join_query_term
;

non_join_query_primary:
simple_table
;

simple_table:
query_specification
|table_value_constructor
;

table_value_constructor:
VALUES table_value_constructor_list
;

table_value_constructor_list:
row_value_constructor
|table_value_constructor_list COMMA row_value_constructor
;

query_specification:
SELECT select_list from_clause wc gbc hc
```

```
//meaning wc:[ <where clause> ], gbc:[ <group by clause> ] hc[ <having clause> ]
;

table_expression:
from_clause wc gbc hc
//meaning wc:[ <where clause> ], gbc:[ <group by clause> ] hc[ <having clause> ]
;

from_clause :
FROM table_reference_list
;

table_reference :
table_name
|table_name correlation_specification
|table_name AS correlation_specification
;

correlation_specification :
 correlation_name
;

table_reference_list:
table_reference
|table_reference_list COMMA table_reference
;

wc:
/*empty*/
| WHERE search_condition
;

gbc:
//empty
|GROUP BY grouping_column_reference_list
;

hc:
//empty
|HAVING search_condition
;

grouping_column_reference_list:
grouping_column_reference
|grouping_column_reference COMMA grouping_column_reference_list
;
```

```
grouping_column_reference :
column_reference
;

select_list :
ASTERISK
|select_list2
;

select_list2:
select_sublist
|select_list2 COMMA select_sublist
;

select_sublist:
derived_column
| qualifier PERIOD ASTERISK
;

derived_column :
value_expression
|value_expression as_clause
;

as_clause :
column_name
|AS column_name
;

value_expression:
numeric_value_expression
;

numeric_value_expression :
term
|   numeric_value_expression PLUS term
|   numeric_value_expression MINUS term
;

term :
factor
|   term ASTERISK factor
|   term SOLIDUS factor
;
```

```
factor :
numeric_primary
|sign numeric_primary
;

sign :
PLUS
|MINUS
;

numeric_primary:
value_expression_primary
;

unsigned_value_specification:
unsigned_literal
;

unsigned_literal:
unsigned_numeric_literal
;

unsigned_numeric_literal:
exact_numeric_literal
;
exact_numeric_literal:
unsigned_integer
;

unsigned_integer:
UNSIGNED_INT
;

value_expression_primary:
unsigned_value_specification
| column_reference
| LP value_expression RP
;

column_reference:
qualifier PERIOD column_name
|column_name
;

qualifier:
table_name
```

```
;


table_name:
identifier
;

column_name:
identifier
;

correlation_name:
identifier
;

search_condition:
boolean_term
|search_condition OR boolean_term
;

boolean_term :
boolean_factor
|boolean_term AND boolean_factor
;

boolean_factor:
NOT boolean_test
|boolean_test
;

boolean_test :
boolean_primary
;

boolean_primary :
predicate
| LP search_condition RP
;

predicate :
comparison_predicate
       | exists_predicate
;

truth_value:
TRUET
```

```
|FALSET
|UNKNOWN
;

comparison_predicate:
 row_value_constructor comp_op row_value_constructor
;

comp_op: //comparisson operators
EQUALOP
|   LTOP GTOP// meaning not equals operator <>
|   LTOP //less than
|   GTOP //greater than
|   LTOP EQUALOP //less than or equal
|   GTOP EQUALOP // greater than or equal
;

row_value_constructor :
row_value_constructor_element
;

row_value_constructor_element :
value_expression
|   NULLT // meaning null_specification
|   DEFAULT // meaning default_specification
;

exists_predicate:
EXISTS subquery
;

subquery :
LP query_expression RP
;

table_definition:
create_table glt table_name LP table_element_list RP on_comit
;

create_table:
 CREATE TABLE
;

identifier:
actual_identifier
;
```

```
actual_identifier:
IDENTIFIER
;

table_element_list:
table_element
| table_element_list COMMA table_element
;

table_element:
column_definition
| table_constraint_definition
;

table_constraint_definition:
table_constraint
| constraint_name_definition table_constraint
;

table_constraint:
unique_constraint_definition
|referential_constraint_definition
|check_constraint_definition
;

referential_constraint_definition :
FOREIGN KEY LP reference_column_list RP references_specification
;

reference_column_list:
column_name_list
;

references_specification:
REFERENCES referenced_table_and_columns
;

referenced_table_and_columns :
table_name
|table_name LP reference_column_list RP
;
unique_constraint_definition:
unique_specification LP unique_column_list RP
;
```

```
unique_specification : UNIQUE
| PRIMARY KEY
;

unique_column_list :
column_name_list
;

column_name_list:
 column_name
| column_name_list COMMA column_name
;

constraint_name_definition:
CONSTRAINT identifier
;

check_constraint_definition:
 CHECK LP search_condition RP
;

constraint_name :
qualified_name
;

qualified_name :
 qualified_identifier
;

qualified_identifier :
identifier
;

schema_name:
catalog_name PERIOD unqualified_schema_name
|unqualified_schema_name
;

catalog_name:
identifier
;

unqualified_schema_name :
identifier
;
```

```
column_definition:
column_name data_type
;

data_type:
numeric_type
;

numeric_type:
exact_numeric_type
;

exact_numeric_type:
INTEGERT
|INT
;
```

Apéndice B

Resultados de ejecución

B.1. Fichero “input.sql”

```
create table player {  id int primary key  };

create table board {
  int x,
  int y,
  int id,
  primary key x,y;
  foreign key(id) references player(id);
 };

create view nowPlaying(id) as
  select p.id
  from player p
  where exists (select b.id from board b where b.id=p.id);

create view checked(id) as
  select p.id
  from player p
  where exists (select n.id from nowPlaying n where n.id = p.id) and
        not exists (select b1.id from board b1
                    where b1.id = p.id and
                          not exists (select b2.id from board b2
                                      where (b2.x - b1.x) * (b2.y-b1.y)=0
                                            and (b1.id <> b2.id)));
```

B.2. Estructura “C++”

```
SCHEMA:

TABLE: board
COLUMNS:
<[NAME: x, TYPE: INTEGER],[NAME: y, TYPE: INTEGER],[NAME: id, TYPE: INTEGER]>
TABLE CONSTRAINTS:
<[PRIMARY KEY: <[Name: x],[Name: y]>],[FOREIGN KEY: <[Name: id]> REFERENCES
<[Name: id]> OF player]>
```

```
ROWS = 2

VIEW: checked
COLUMNS:
<[Name: id]>
QUERY:{
  SELECT : (p.id)
  FROM: [<player as p>]
  WHERE : (EXISTS ({SELECT : (n.id)
    FROM: [<nowPlaying as n>]
    WHERE : (n.id == p.id)
    GROUP BY :
    HAVING :
  }) and (not EXISTS ({SELECT : (b1.x,b1.y,b1.id)
FROM: [<board as b1>]
WHERE : ((b1.id == p.id) and
(not EXISTS ({SELECT : (b2.x,b2.y,b2.id)
      FROM: [<board as b2>]
      WHERE : ((((b2.x - b1.x) * (b2.y - b1.y)) == 0)
      and (b1.id != b2.id))
      GROUP BY :
      HAVING :
      })))
GROUP BY :
HAVING :
})))
  GROUP BY :
  HAVING :
}

VIEW: nowPlaying
COLUMNS:
<[Name: id]>
QUERY:{
  SELECT : (p.id)
  FROM: [<player as p>]
  WHERE : EXISTS ({SELECT : (b.x,b.y,b.id)
    FROM: [<board as b>]
    WHERE : (b.id == p.id)
    GROUP BY :
    HAVING :
  })
  GROUP BY :
  HAVING :
}

TABLE: player
COLUMNS:
<[NAME: id, TYPE: INTEGER]>
TABLE CONSTRAINTS:
<[PRIMARY KEY: <[Name: id]>]>
ROWS = 2
```

B.3. Fórmulas de restricciones del ejemplo

B.3.1. Fórmula de restricciones para la tabla `player`

```
#(player)=
{(((true AND (true AND (player.id.0 != player.id.1))) AND true),
{player.id -> player.id.0}),

(((true AND (true AND (player.id.0 != player.id.1))) AND true),
{player.id -> player.id.1})}
```

B.3.2. Fórmula de restricciones para la tabla `board`

```
#(board)=
{((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)),
{board.x -> board.x.0,board.y -> board.y.0,board.id -> board.id.0}),

(((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)),
{board.x -> board.x.1,board.y -> board.y.1,board.id -> board.id.1})}
```

B.3.3. Fórmula de restricciones para la vista `nowPlaying`

```
#(nowPlaying)=
{(((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND (board.id.0 == player.id.0))
OR (((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND
(board.id.1 == player.id.0)))),
{nowPlaying.id -> player.id.0}),

((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND (board.id.0 == player.id.1))
OR (((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND
(board.id.1 == player.id.1)))),{nowPlaying.id -> player.id.1})}
```

B.3.4. Fórmula de restricciones para la vista `checked`

```
#(checked)=
{((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND (board.id.0 == player.id.0))
OR (((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND
(board.id.1 == player.id.0)))) AND (player.id.0 == player.id.0)) OR
((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND (board.id.0 == player.id.1))
OR (((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND
(board.id.1 == player.id.1)))) AND (player.id.1 == player.id.0)))
and (not ((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1))
AND ((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND ((board.id.0 == player.id.0)
AND (not ((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1))
AND ((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND
((((board.x.0 - board.x.0) * (board.y.0 - board.y.0)) == 0) and
(board.id.0 != board.id.0))) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((((board.x.1 - board.x.0) * (board.y.1 - board.y.0)) == 0) and
(board.id.0 != board.id.1)))))))) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((board.id.1 == player.id.0) and (not ((((((board.id.0 == player.id.0)
OR (board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((((board.x.0 - board.x.1) * (board.y.0 - board.y.1)) == 0) and
(board.id.1 != board.id.0))) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((((board.x.1 - board.x.1) * (board.y.1 - board.y.1)) == 0) and
(board.id.1 != board.id.1)))))))))),
{checked.id -> player.id.0}),

((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((true AND (true AND (player.id.0 != player.id.1))) AND true) AND
((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
```

```
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND (board.id.0 == player.id.0))
OR (((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND
((true AND (true AND ((board.x.0 != board.x.1) OR
(board.y.0 != board.y.1)))) AND true)) AND (board.id.1 == player.id.0))))
AND (player.id.0 == player.id.1)) OR ((((true AND (true AND
(player.id.0 != player.id.1))) AND true) AND
((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND ((true AND
(true AND ((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true))
AND (board.id.0 == player.id.1)) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
(board.id.1 == player.id.1)))) AND (player.id.1 == player.id.1))) and
(not ((((((board.id.0 == player.id.0) OR (board.id.0 == player.id.1)) AND
((board.id.1 == player.id.0) OR (board.id.1 == player.id.1))) AND ((true AND
(true AND ((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true))
AND ((board.id.0 == player.id.1) and (not ((((((board.id.0 == player.id.0)
OR (board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true))
AND ((((board.x.0 - board.x.0) * (board.y.0 - board.y.0)) == 0) and
(board.id.0 != board.id.0))) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((((board.x.1 - board.x.0) * (board.y.1 - board.y.0)) == 0) and
(board.id.0 != board.id.1))))))) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((board.id.1 == player.id.1) and (not ((((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((((board.x.0 - board.x.1) * (board.y.0 - board.y.1)) == 0) and
(board.id.1 != board.id.0))) OR (((((board.id.0 == player.id.0) OR
(board.id.0 == player.id.1)) AND ((board.id.1 == player.id.0) OR
(board.id.1 == player.id.1))) AND ((true AND (true AND
((board.x.0 != board.x.1) OR (board.y.0 != board.y.1)))) AND true)) AND
((((board.x.1 - board.x.1) * (board.y.1 - board.y.1)) == 0) and
(board.id.1 != board.id.1))))))))))),
{checked.id -> player.id.1})}
```

B.4. Fichero "output.mzn"

```
%%%%% Declaración de variables: %%%%%

var 1..5: board_id_0;
var 1..5: board_id_1;
var 1..5: board_x_0;
```

```
var 1..5: board_x_1;
var 1..5: board_y_0;
var 1..5: board_y_1;
var 1..5: player_id_0;
var 1..5: player_id_1;

%%%%% Definición de restricciones: %%%%%

constraint ((((true /\ (true /\ (player_id_0 != player_id_1))) /\ true)
/\ ((((((true /\ (true /\ (player_id_0 != player_id_1))) /\ true) /\
((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\
(true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true))
/\ (board_id_0 = player_id_0)) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1)
\/ (board_y_0 != board_y_1)))) /\ true)) /\ (board_id_1 = player_id_0)))) /\
(player_id_0 = player_id_0)) \/ ((((true /\ (true /\ (player_id_0 != player_id_1)))
/\ true) /\ ((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\ (true /\
((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)) /\
(board_id_0 = player_id_1)) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1)
\/ (board_y_0 != board_y_1)))) /\ true)) /\ (board_id_1 = player_id_1)))) /\
(player_id_1 = player_id_0))) /\ (not ((((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1) \/
(board_y_0 != board_y_1)))) /\ true)) /\ ((board_id_0 = player_id_0) /\
(not ((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\ (true /\
((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_0 - board_x_0) * (board_y_0 - board_y_0)) = 0) /\
(board_id_0 != board_id_0))) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1) \/
(board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_1 - board_x_0) * (board_y_1 - board_y_0)) = 0) /\
(board_id_0 != board_id_1))))))) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1) \/
(board_y_0 != board_y_1)))) /\ true)) /\ ((board_id_1 = player_id_0) /\
(not ((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\
(true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_0 - board_x_1) * (board_y_0 - board_y_1)) = 0) /\
(board_id_1 != board_id_0))) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1)
\/ (board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_1 - board_x_1) * (board_y_1 - board_y_1)) = 0) /\
(board_id_1 != board_id_1)))))))))) \/ (((true /\ (true /\
(player_id_0 != player_id_1))) /\ true) /\ ((((((true /\ (true /\
(player_id_0 != player_id_1))) /\ true) /\ ((((((board_id_0 = player_id_0)
\/ (board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1)
\/ (board_y_0 != board_y_1)))) /\ true)) /\ (board_id_0 = player_id_0)) \/
```

```
(((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\
(true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true))
/\ (board_id_1 = player_id_0)))) /\ (player_id_0 = player_id_1)) \/
((((true /\ (true /\ (player_id_0 != player_id_1))) /\ true) /\
((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\ (true /\
((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)) /\
(board_id_0 = player_id_1)) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1) \/
(board_y_0 != board_y_1)))) /\ true)) /\ (board_id_1 = player_id_1)))) /\
(player_id_1 = player_id_1))) /\ (not ((((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1) \/
(board_y_0 != board_y_1)))) /\ true)) /\ ((board_id_0 = player_id_1) /\
(not ((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\
(true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_0 - board_x_0) * (board_y_0 - board_y_0)) = 0) /\
(board_id_0 != board_id_0))) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1)
\/ (board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_1 - board_x_0) * (board_y_1 - board_y_0)) = 0) /\
(board_id_0 != board_id_1))))))) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1)
\/ (board_y_0 != board_y_1)))) /\ true)) /\ ((board_id_1 = player_id_1) /\
(not ((((((board_id_0 = player_id_0) \/ (board_id_0 = player_id_1)) /\
((board_id_1 = player_id_0) \/ (board_id_1 = player_id_1))) /\ ((true /\
(true /\ ((board_x_0 != board_x_1) \/ (board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_0 - board_x_1) * (board_y_0 - board_y_1)) = 0) /\
(board_id_1 != board_id_0))) \/ (((((board_id_0 = player_id_0) \/
(board_id_0 = player_id_1)) /\ ((board_id_1 = player_id_0) \/
(board_id_1 = player_id_1))) /\ ((true /\ (true /\ ((board_x_0 != board_x_1) \/
(board_y_0 != board_y_1)))) /\ true)) /\
((((board_x_1 - board_x_1) * (board_y_1 - board_y_1)) = 0) /\
(board_id_1 != board_id_1))))))))))));

%%%%% Llamada al resolutor: %%%%%

solve satisfy;

%%%%% Formato de salida: %%%%%

output [" INSERT INTO board (id,x,y) VALUES (", show(board_id_0),",",
show(board_x_0),",", show(board_y_0),");"," INSERT INTO board (id,x,y)
 VALUES (", show(board_id_1),",", show(board_x_1),",", show(board_y_1),
");"," INSERT INTO player (id) VALUES (", show(player_id_0),");","
INSERT INTO player (id) VALUES (", show(player_id_1),");"];
```

Apéndice C

Manual de usuario

STCG es una generador de casos de prueba para relaciones SQL.

Para poder utilizarlo primero has de instalarlo siguiendo los pasos de la sección de instalación.

Si ya lo tienes instalado puedes ver la sección de ejecución para consultar las opciones y funcionalidades de STCG.

C.1. Instalación

C.1.1. Hazte con la herramienta

- Puedes descargarte la última versión de STCG ejecutable en linux en la siguiente página: `http://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/STCG`
- Si lo prefieres puedes obtener el código del proyecto desde el repositorio de subversion del proyecto: `svn checkout http://stcg.googlecode.com/svn/trunk/ stcg-read-only`

  Para compilarlo en linux:

  - Muévete a la carpeta de STCG.
  - Haz `make; sudo make install` en la consola de comandos.

C.1.2. También puedes necesitar

- STCG genera código *MiniZinc* [6] para representar las restricciones que deben satisfacer los casos de prueba buscados. Para poder obtener resultados necesitas ejecutar MiniZinc, podrás encontrarlo en su página web: `http://www.g12.csse.unimelb.edu.au/MiniZinc/`
- Si lo que quieres es *compilar* el código también necesitarás:
  - *GNU Bison*: `http://www.gnu.org/software/bison/`

- *Flex*: `http://flex.sourceforge.net/`
- *g++* u otro compilador de C++ a tu elección.

C.2. Ejecución

Si ya tienes instalado STCG podrás ejecutarlo desde la consola de comandos.

C.2.1. Obtener ayuda y forma de uso

Puedes obtener ayuda directamente de la herramienta ejecutando `stcg --help` o `stcg -?` Esto indicará las opciones disponibles y su significado.

También puedes ejecutar `stcg --usage` para obtener una breve descripción de la forma de uso de la herramienta.

C.2.2. Obtener casos de prueba

STCG obtiene casos de prueba para una determinada relación SQL, para ello necesita:

- La descripción de la base de datos a estudiar, o al menos de las vistas y tablas relacionadas con la relación objetivo, a través del código SQL que las describe. Esto puede hacerse de dos formas:
  - Ejecutando STCG con la opción –input o -i indicando el nombre del archivo con el código SQL: `stcg --input entrada.sql` o `stcg -i entrada.sql`
  - Escribiendo desde la entrada de texto de la consola el código SQL que define las relaciones que se quieren estudiar. Para terminar la entrada de datos hay que indicar el final de fichero. Esto puede hacerse mediante la combinación Ctrl+D.
- el nombre de la tabla o vista para la cual se quiere obtener el caso de prueba. Esto puede especificarse de dos maneras:
  - Utilizando la opción –formula o -f: `stcg --formula Nombre-Relacion` o `stcg -f Nombre-Relacion`
  - Si no se utilizó la opción -f la herramienta preguntará el nombre de la relación una vez procesada la entrada de código SQL que define la base de datos.
- El máximo número de filas por cada tabla implicada en el caso de prueba. La herramienta preguntará al usuario estos datos.

Una vez que la herramienta dispone de estos datos proporcionará el caso de prueba en forma de código MiniZinc. Dependiendo de las opciones indicadas la salida tendrá las siguientes características:

- Si se especifica la opción –output salida.mzn o -o salida.mzn la salida se escribirá en el archivo especificado: `stcg --output salida.mzn` o `stcg -o salida.mzn`. En caso contrario la salida se efectuará por la salida estándar (usualmente la consola de comandos).
- Si se especifica la opción –verbose o -v la salida contendrá, además del código MiniZinc, la estructura de la base de datos inferida a través del análisis del código SQL que la define: `stcg --vervose` o `stcg -v`

Para obtener el caso de prueba hay que resolver el modelo en código MiniZinc que proporciona STCG. Se puede hacer esto de las siguientes maneras:

- Ejecutando directamente el resolutor de MiniZinc sobre el fichero que contiene el código MiniZinc: `MiniZinc salida.mzn`
- Transformando el código MiniZinc a código FlatZinc para resolverlo en cualquiera de los resolutores que lo implementan. `mzn2fzn salida.mzn` para transformarlo al archivo salida.fzn, `fz salida.fzn` para resolverlo, en este caso con el resolutor de FlatZinc que implementa Gecode[8] (requiere la instalación de Gecode).

C.2.3. Todo en uno

Presentamos el siguiente ejemplo para ejecutar los pasos anteriores con una sola línea en la consola de comandos:

```
 cat entrada.sql >> salida.sql; stcg -i entrada.sql -o salida.mzn;
MiniZinc salida.mzn >> salida.sql
```

Con esto conseguimos un archivo (salida.sql) que conforma el caso de prueba buscado incluyendo la definición de la base de datos y la inserción del caso de prueba obtenido en la misma.

C.2.4. Dudas, sugerencias, problemas...

Puede contactar con los desarrolladores de la herramienta para solventar dudas, hacer proposiciones respecto a la herramienta o comunicar comportamientos inesperados del programa. Si deseas comunicar un problema de la herramienta, por favor, incluye una explicación detallada de los pasos que reproducen el problema y toda la información adicional que pudiera ser de utilidad.

Contacto: jose.11.10.88(arroba)gmail.com y senrof21(arroba)gmail.com

Apéndice D

Ejemplos de Equivalencia entre sentencias SQL

Este apéndice muestra la equivalencia entre algunas de las sentencias SQL no soportadas directamente por la herramienta con otras que sí lo están. No se pretende demostrar estos resultados sino ilustrar estas equivalencias mediante ejemplos y explicaciones informales. Algunos de los ejemplos son adaptaciones de los encontrados en el MySQL 5.0 Reference Manual [14]

- *Subconsultas en la cláusula* `FROM`*:* Las subconsultas en la cláusula `FROM` pueden reescribirse como vistas SQL, veamos el siguiente ejemplo, que puede generalizarse a otras situaciones:

  considérese la tabla `CREATE TABLE t1 (s1 INT, s2 INT, s3 INT);`

  Es equivalente la consulta con subconsulta en la cláusula `FROM`:

  ```
  SELECT sb1,sb2,sb3
  FROM (SELECT s1 AS sb1, s2 AS sb2, s3*2 AS sb3 FROM t1)
  AS sb WHERE sb1 >1;
  ```

  y la siguiente creación de vista y consulta:

  ```
  CREATE VIEW sb AS
  SELECT s1 AS sb1, s2 AS sb2, s3*2 AS sb3 FROM t1;
  SELECT sb1,sb2,sb3 FROM sb;
  ```

- *Subconsultas con* `ANY`*:*

  Las subconsultas con el operador `ANY` son de la forma `operando <operador de comparación>ANY (subconsulta)` y son `true` si la comparación es cierta para cualquiera de los valores que devuelve la subconsulta.

  Así, podemos establecer que es equivalente la consulta

`SELECT s1 FROM t1 WHERE s1 op ANY (SELECT s2 FROM t2);` Donde `op` es un operador de comparación.

y la consulta con subconsulta existencial

```
SELECT s1 FROM t1 WHERE EXISTS (SELECT s2 FROM t2 where s1 op
s2)
```

- *Subconsultas con* `IN`*:*

  El operador `IN` es equivalente a `= ANY` [14] por lo que se puede aplicar la explicación anterior.

- *Subconsultas con* `ALL`*:*

  Las consultas con el operador `ALL` siguen la siguiente sintaxis: `operando <operador de comparación>ANY (subconsulta)` y son `true` cuando la comparación es cierta para todos los valores devueltos por la subconsulta, es decir, no existe un valor devuelto por la subconsulta para el cual la comparación sea falsa.

  Con esta semántica podemos establecer que:

  `SELECT s1 FROM t1 WHERE s1 op ALL (SELECT s2 FROM t2);` Donde `op` es un operador de comparación.

  es equivalente a:

  `SELECT s1 FROM t1 WHERE NOT EXISTS (SELECT s2 FROM t2 WHERE s1 !op s2 );` Donde `!op` es la negación del operador de comparación.

- *Consultas con* `NOT IN`*:*

  `NOT IN` es una forma equivalente a `<>ALL` por lo que el apartado anterior puede considerarse una forma general de afrontar este caso particular.

Índice alfabético

Bibliografía

[1] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "Declarative Debugging of Wrong and Missing Answers for SQL Views," in *Eleventh International Symposium on Functional and Logic Programming (FLOPS 2012)*, ser. LNCS, vol. 7294. Springer-Verlag, 2012.

[2] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "Full predicate coverage for testing sql database queries," *Softw. Test. Verif. Reliab.*, vol. 20, no. 3, pp. 237–288, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1002/stvr.v20:3

[3] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez, "Applying constraint logic programming to sql test case generation," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, M. Blume, N. Kobayashi, and G. Vidal, Eds. Springer Berlin / Heidelberg, 2010, vol. 6009, pp. 191–206.

[4] K. Marriott and P. J. Stuckey, *Programming with Constraints: An Introduction*. MIT Press, 1998.

[5] R. Caballero, J. Luzon-Martin, and A. Tenorio, "Test-case generation for sql nested queries with existential conditions," in *Jornadas Sistedes 2012 JISBD; PROLE; JCIS*. Electronic Communications of the European Association of Software Science and Technology, 2012, in Press.

[6] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language." ser. Lecture Notes in Computer Science, C. Bessiere, Ed., vol. 4741. Springer, 2007, pp. 529–543.

[7] D. L. Williams, *A (Partial) Introduction to Software Engineering Practices and Methods*, 2008-09, ch. White-Box Testing, nCSU CSC326 Course Pack, 2008-2009 (Fifth) Editions. [Online]. Available: http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf

[8] C. Schulte, M. Z. Lagerkvist, and G. Tack, "Gecode." [Online]. Available: http://www.gecode.org/

[9] J. Leffler, "Bnf grammar for iso/iec 9075:1992 - database language sql (sql-92)." [Online]. Available: http://savage.net.au/SQL/sql-92.bnf

[10] T. F. Project, "flex: The Fast Lexical Analyzer." [Online]. Available: http://flex.sourceforge.net/

[11] Various Contributors and the GNU Project, *Bison - GNU parser generator*. Free Software Foundation, Inc., 1998-2012. [Online]. Available: http://www.gnu.org/software/bison/

[12] I. Salcedo-Ramos and A. Tenorio-Fornés, "Uso de restricciones para modelar funciones sobre strings en SQL," 2012. [Online]. Available: http://subversion.assembla.com/svn/programaci-n-declarativa-avanzada/trunk/practica2/practica2PDA.mzn

[13] J. Tuya, M. J. Suárez-Cabal, and C. de la Riva, "A practical guide to sql white-box testing," *SIGPLAN Not.*, vol. 41, no. 4, pp. 36–41, Apr. 2006. [Online]. Available: http://doi.acm.org/10.1145/1147214.1147221

[14] M. R. Manual, "Mysql 5.0 reference manual," *Syntax*, 2010. [Online]. Available: http://dev.mysql.com/doc/refman/5.6/en/index.html